Carl von Ossietzky Universität Oldenburg
Fachbereich Informatik
Abteilung Informationssysteme
Prof. Dr. H.-J. Appelrath

DIPLOMARBEIT

3D Visualisierungskonzepte für virtuelle Labore

vorgelegt von:

Wilko Heuten

Erstprüfer:

Prof. Dr. H.-J. Appelrath

Zweitprüfer:

Dipl.-Inform. D. Boles

Oldenburg, den 30. April 2001

Danksagung

Ich danke allen, die mich bei der Erstellung dieser Arbeit unterstützt haben. Mein ganz besonderer Dank gilt Marco Schlattmann für die hervorragende Betreuung nicht nur während dieser sechs Monate, sondern auch während meiner dreijährigen Tätigkeit als Hilfswissenschaftler im Projekt GENLAB.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 1

1.1.1 Probleme der zweidimensionalen Darstellung 1

1.1.2 3D Visualisierung 6

1.2 Zielsetzung 7

1.3 Gliederung 8

2 Problemanalyse und Anforderungen **9**

2.1 Allgemeine Eigenschaften und Anforderungen virtueller Labore . . . 9

2.2 Berechnung und Darstellung der 3D-Szene 11

2.2.1 Ermittlung sichtbarer Flächen 12

2.2.2 Beleuchtung 13

2.2.3 Schattierung 13

2.2.4 Reflexionseigenschaften 14

2.2.5 Transparenz 14

2.2.6 Schatten 14

2.2.7 Berücksichtigung von Materialeigenschaften / Texturen 15

2.2.8 Dreidimensionale Darstellung / Projektion 15

2.2.9 Hardwareunterstützung 16

2.2.10 Zusammenfassung 16

2.3 Interaktion und Manipulation der 3D-Objekte 17

2.3.1 Positionierung 18

2.3.2 Rotation 18

2.3.3 Auswahl 18

2.3.4 Kollisionsentdeckung 18
2.3.5 Navigation 20
2.3.6 Benutzer-System Schnittstelle 20
2.4 Erstellung der 3D-Modelle 22
2.4.1 Modellierungstechniken 22
2.4.2 Beschaffenheit der Modelle 23
2.5 Datenhaltung und Zugriff 23
2.6 3D-Grafikformate 24
2.7 Zusammenfassung 25

3 Analyse der Entwicklungswerkzeuge 27
3.1 Definitionen und Begriffserläuterungen 27
3.2 Kriterien zur Wahl der Entwicklungswerkzeuge 30
3.3 Analyse 32
3.3.1 Die Autorensysteme im Detail 32
3.3.2 Die Programmiersprachen und Bibliotheken im Detail 38
3.3.3 Beschreibungssprachen, sonstige Tools 40
3.3.4 Ergebnis der Analyse 43
3.4 Zusammenfassung 43

4 Entwurf der 3D-Visualisierung 45
4.1 Entwurf: Teil I 45
4.1.1 Identifikation der Komponenten 46
4.1.2 Schnittstellenlayer 47
4.1.3 Schnittstellenklasse `3DView` 51
4.1.4 Schnittstellenklasse `Camera` 56
4.1.5 Schnittstellenklasse `Transformnode` 57
4.1.6 Schnittstellenklasse `World` 58
4.1.7 Schnittstellenklasse `Universe` 58
4.2 Entwurf: Teil II 60
4.2.1 Aufbau des GENLAB-Frameworks 60

4.2.2 Anbindung des Viewports / 3D DREAMS an das GENLAB-Framework 63
4.2.3 Anpassen und Erweitern des GENLAB-Frameworks für die 3D-Visualisierung 64
4.2.4 Bausteinkomponenten 75
4.3 Zusammenfassung 75

5 Implementierungsaspekte 79

5.1 Implementierungsumgebung 79
5.2 Erzeugung manipulierbarer und interaktiver dreidimensionaler Objekte 80
5.3 Weiterleitung der Anwendereingaben an 3D-Objekte 84
5.4 Interaktive Transformation eines 3D-Objektes 86
5.5 Interaktionen zwischen Views 91
5.6 Bausteinkomponenten 95
5.6.1 Schalter 96
5.6.2 Dynamische Anzeigen 97
5.6.3 Virtueller Arbeitsbereich 99
5.7 Performancesteigernde Maßnahmen 99
5.7.1 Darstellung von Arbeitsflächen und Wänden 99
5.7.2 Wireframedarstellung beim Bewegen von 3D-Objekten 100
5.8 Erweiterung und Korrekturen von 3D DREAMS 102
5.8.1 Kollisionen, Objektannäherung 102
5.8.2 Erzeugung von Objekten in der Runtime-Umgebung 103
5.9 Zusammenfassung 104

6 Erstellung eines prototypischen Labors 105

6.1 Installationshinweise / Vorbereitung 105
6.2 Konstruktion der 3D-Modelle 106
6.2.1 Anforderungen an 3D-Modelle für den interaktiven Gebrauch 107
6.2.2 Modellierungstechniken 108
6.2.3 Konkrete Modellierung einer Mikrowelle 108
6.3 Import der 3D-Modelle 110

6.4 Erstellen der View und Controllerklassen 111
6.5 Anbindung der Viewobjekte an die 3D-Modelle 116
6.6 Konfiguration der Szene 116
6.7 Moduldefinitionen 117
6.8 Distribution 118

7 Bewertung und Ausblick 121

A Listing zur Erstellung einer 3D-Szene mit dem Mathengine-Toolkit 125

Glossar 129

Literaturverzeichnis 133

Index 135

Versicherung 137

Abbildungsverzeichnis

1.1 Die Experimentierumgebung in GENLAB 2

1.2 Erstellung eines Views im 2D-Labor 3

1.3 Verschiedene Repräsentationen einer Komponente 3

1.4 Verschiedene Repräsentationen im 2D-Labor setzen sich aus unterschiedlichen Quellen zusammen 5

1.5 Im 3D-Labor greifen alle Repräsentationen auf die gleiche Quelle zurück 6

2.1 Zwei verschiedene Projektionen der selben Linie. (a) Zentralprojektion, (b) Parallelprojektion. 16

2.2 Ungenauigkeiten bei der Kollisionsentdeckung mit Bounding Boxes . 19

2.3 Annäherung einer linearen Struktur durch ausgerichtete Bounding-Boxes. Aus [LMCG]. 19

2.4 V-COLLIDE erkennt die Situationen richtig: (a) und (c) sind Kollisionen, (b) ist keine. Aus [LMCG] 20

2.5 Aus Polygonen zusammengesetzte Kugeln: (a) = 72 Polygone, (b) = 288 Polygone und (c) = 1152 Polygone 23

3.1 Aufgaben bei der Erstellung viruteller Labore 29

3.2 3D DREAMS in Director 8 integriert 35

3.3 Programmierschnittstellen zu VRML-Szenen. Aus [KRSD98]. 42

4.1 Die Komponenten für die 3D-Visualisierung eines virtuellen Labors . 47

4.2 Hierarchische Objektstruktur einer VRML-Szene 48

4.3 Hierarchische Schnittstelle mit Mehrfachvererbung 50

4.4 Bei Einfachvererbung können keine Routinen in eine Oberklasse ausgelagert werden 51

4.5 Darstellung des Schnittstellenlayers 52

4.6 Schnittstellenlayer zwischen dem Laborkern, der Render-Engine und der Kollisionsentdeckung 59

4.7 Komposition eines Views durch mehrere Controller im GENLAB-Framework 61

4.8 Die Basisklassen des GENLAB-Frameworks 62

4.9 Das GENLAB-Framework wird durch diese abstrakten View3D-Klassen zur Erstellung von echten 3D-Objekten erweitert 69

4.10 Erweiterung des GENLAB-Frameworks um abstrakte Klassen zur Erstellung neuer Module mit 3D-Untersützung 74

4.11 Übersicht über die neue Controller-Struktur 76

4.12 Übersicht über die neue View-Struktur 77

5.1 Hierarchie einer Szene in 3D DREAMS 81

5.2 Viewhierarchie der Mikrowelle 82

5.3 Zuweisung eines View-Skriptes an ein 3D-Modell 83

5.4 Ablauf der Methode `mMouseDown` in der Controller-Basisklasse `3DCTRL` 88

5.5 Interaktionsbeziehungen 91

5.6 Verwendung einer Hintergrundbitmap, um den Eindruck einer Arbeitsumgebung zu vermitteln 100

5.7 Wireframe-Modus beim Drag der Mikrowelle 101

6.1 Workflow zur Erstellung eines Versuches mit 3D-Integration 106

6.2 Die Konstruktion eines 3D-Modells am Beispiel einer Mikrowelle . . 108

6.3 Verbinden zweier 3D-Objekte 109

6.4 Hierarchie des Mikrowellen-Modells 110

6.5 Hierarchie der Mikrowelle nach dem Import in 3D DREAMS 111

6.6 Geänderte Hierarchie der Mikrowelle in 3D DREAMS 111

1 Einleitung

1.1 Motivation

Im Oldenburger Forschungs- und Entwicklungsinstitut für Informatik-Werkzeuge und -Systeme (OFFIS) [OFF00] – Arbeitsgruppe Multimedia-Systeme von Prof. Dr. H.-J. Appelrath – wird seit September 1997 ein Multimediales Gentechnisches Praktikum (GENLAB) [Gen00], [O+97] entwickelt, das das Durchführen gentechnischer Experimente in einer virtuellen Umgebung gestattet. An GENLAB schließt sich das Projekt VIRTLAB [AB99] an. Im Rahmen dieses Projektes wird vom konkreten Genlabor abstrahiert und es werden Methoden und Werkzeuge zur Entwicklung allgemeiner virtueller multimedialer naturwissenchaftlich-technischer Labore und Praktika erarbeitet.

In GENLAB wurde bereits ein Framework geschaffen, das die Erstellung gentechnischer Labore stark vereinfacht. Die Anwendung dieses Frameworks ist nicht auf gentechnische Labore beschränkt. Es ist in der objektorientierten Skriptsprache *Lingo* [Eps98] implementiert, die Bestandteil des Multimedia-Autorensystems *Director* der Firma MACROMEDIA ist. Wie Sigurd Elfreich in seiner Diplomarbeit [Elf99] gezeigt hat, lassen sich die im Framework vorgeschlagenen Konzepte auch auf andere OO-Sprachen übertragen und erweitern. Grundlegendes Prinzip des Frameworks und damit zur Erstellung virtueller Labore ist das Model-View-Controller Konzept [G+98], das sich mittlerweile bei der Entwicklung multimedialer Benutzerschnittstellen durchgesetzt hat. Dadurch werden Modelle, ihre Bildschirmrepräsentationen und die Benutzerinteraktionen jeweils von einander gekapselt und so hohe Wiederverwertbarkeit und Änderbarkeit ermöglicht. Die View-Komponente in GENLAB basiert auf der von *Director* zur Verfügung gestellten Spriteverwaltung und ist auf die zweidimensionale Visualisierung der Laborkomponenten beschränkt. Auch Sigurd Elfreich verwendet in seinem Framework ausschließlich zweidimensionale Views. Dies begründet sich auf der einen Seite durch Anforderungen an die Performance, da die Darstellung auf üblichen Multimedia-Rechnern möglich sein soll. Auf der anderen Seite fehlt es an echter 3D-Unterstützung bei den Autorensystemen. Die Wahl der zweidimensionalen Darstellung ist jedoch mit einigen Problemen und Einschränkungen verbunden.

1.1.1 Probleme der zweidimensionalen Darstellung

Mit zweidimensionalen Bildern kann, wie das Projekt GENLAB zeigt (siehe Abbildung 1.1), ein guter räumlichen Effekt erzielt werden. Dieser ist für die Motivation

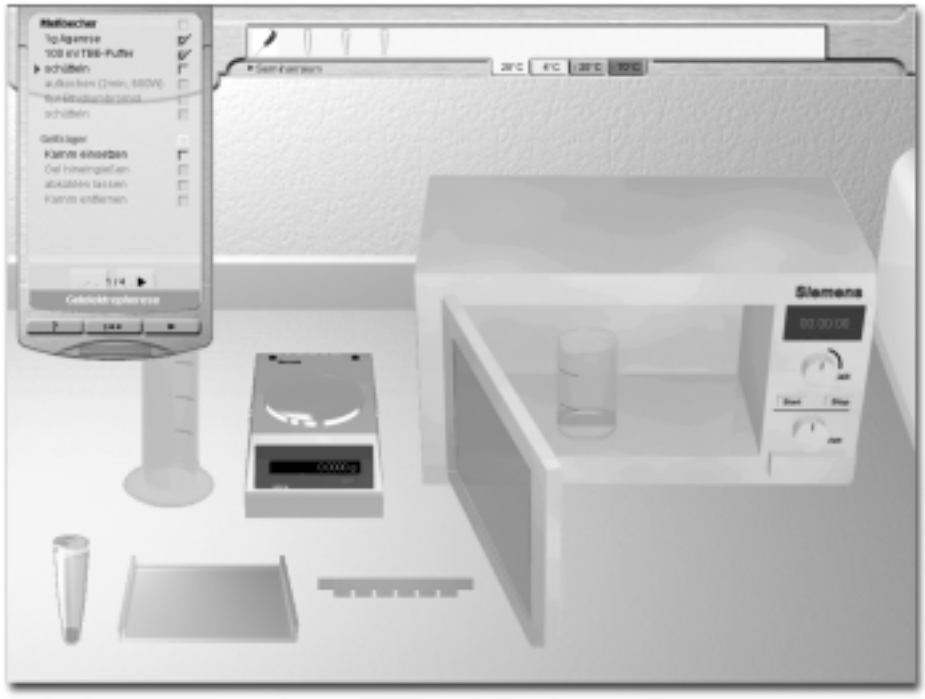

Abbildung 1.1: Die Experimentierumgebung in GENLAB

des Anwenders und dadurch für die Akzeptanz des Produktes ein mitentscheidender Faktor. Die Verwaltung der 2D-Bilder wird üblicherweise über das Sprite-Konzept realisiert. Nach [Tho95] ist ein Sprite ein Bild mit beliebiger Form, das frei positionierbar ist. Sprites können vor oder hinter anderen Sprites liegen, „durchsichtige“ Bereiche enthalten und bewegt werden [...]. Sie können ihre Form verändern, auftauchen, verschwinden und kollidieren. Die Verwendung von Sprites zur Simulation räumlicher Tiefe ergibt folgende Probleme:

Aufwendige Erstellung und Wartung eines interaktiven View-Objektes

Der momentane Arbeitsablauf zur Erstellung eines interaktiven Gerätes ist äußerst aufwendig. Die Komponenten werden in einem 3D-Grafikprogramm konstruiert und texturiert. Dann erfolgt das Setzen der Beleuchtung, die zusammen mit der Textur eine bestimmte Schattierung der Oberfläche ergibt (s. [Heu99]). Das Modell wird in einer Perspektive berechnet, die der ungefähren Betrachtung im Labor entspricht. Das Bild wird in einem 2D-Grafikprogramm weiterverarbeitet und in seine Einzelteile (Schalter, Bedienfelder, Anzeigen etc.) zerlegt (s. Abbildung 1.2). Dieser Vorgang kann sich wiederholen, da z. B. Schalter in mehreren Positionen vorhanden sein müssen. Die einzelnen Bilder werden in die Entwicklungsumgebung importiert und dort wieder sorgfältig pixelgenau zu einem View zusammengesetzt. Die gewählte Perspektive ist statisch. Ist der View einmal importiert, sind die Grenzen einer visuellen (Perspektive, Schattierung) Veränderung bei beweglichen Objekten oder bei einem sich bewegendem Betrachter nur schwer und unbefriedigend durchzuführen. Soll ein Ge-

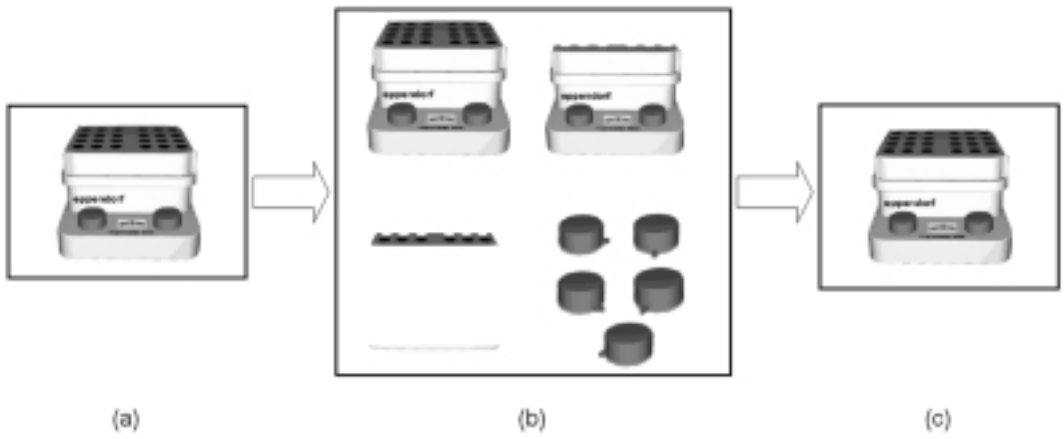

Abbildung 1.2: Erstellung eines Views im 2D-Labor

rät beispielsweise doch an einer anderen Stelle im Labor platziert oder Teile des Views verändert werden, muss der komplette Arbeitsablauf wiederholt werden. Dies bedeutet einen erheblichen Zeitaufwand.

Darstellung verschiedener Repräsentationen eines Views

Oft werden Laborkomponenten auf verschiedene Weise im virtuellen Labor verwendet. In GENLAB gibt es eine Repräsentation für die Experimentierumgebung, einen QUICKTIME-Film zur Ansicht von allen Seiten und ein Photo (vgl. Abbildung 1.3). Zusätzlich vorstellbar sind Animationssequenzen, welche die Bedienung eines Gerätes

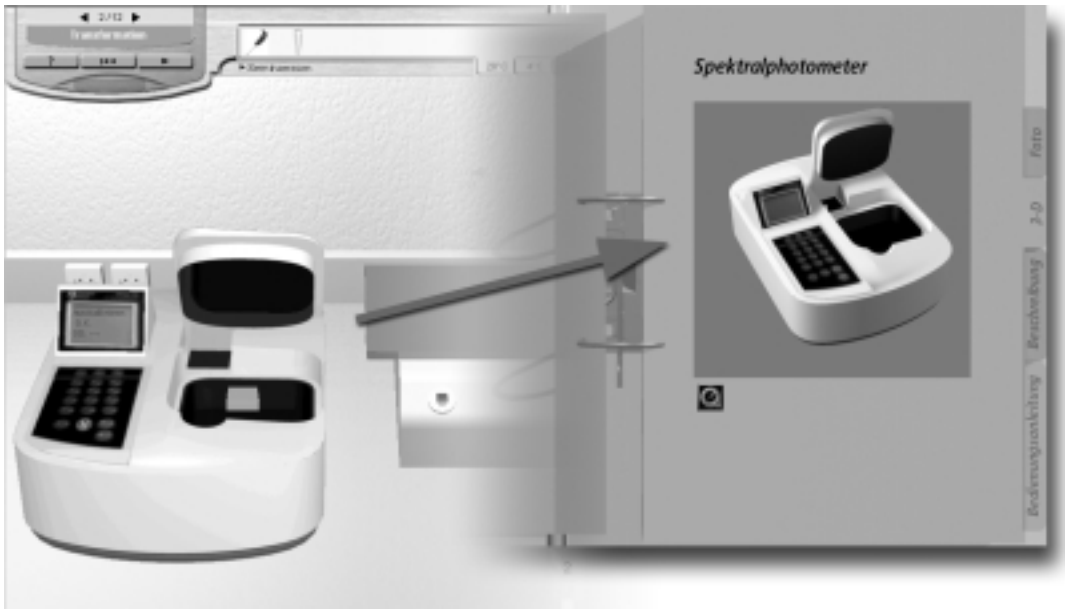

Abbildung 1.3: Verschiedene Repräsentationen einer Komponente

zeigen. Der Arbeitsaufwand zur Erstellung der Repräsentation für die Experimentierumgebung wurde oben beschrieben. Die anderen Repräsentationen lassen sich nahezu automatisch aus dem 3D-Programm erzeugen. Dennoch gibt es dabei zwei Probleme, die durch Abbildung 1.4 veranschaulicht werden:

- ☞ Im Labor werden verschiedene Informationsquellen zur Darstellung unterschiedlicher Repräsentationen zusammengesetzt
- ☞ Einzelne Repräsentationen benötigen oft viele Einzelbilder. Diese erhöhen das Datenvolumen.

Um Inkonsistenzen bezüglich der Darstellung einer Komponente zu vermeiden, müssen selbst bei Veränderung eines kleinen Bestandteils einer Laborkomponente alle anderen Repräsentationen entsprechend geändert werden. Problematisch wird dies, wenn das Produkt schon ausgeliefert wurde und die Aktualisierung „online“ stattfinden muss. Sie wäre nur durch die Übertragung eines oft hohen Datenvolumens durchführbar.

Z-Order Sortierung

Da Sprites an sich nur zweidimensional sind, besitzen sie keine Tiefeninformationen. Um einen dreidimensionalen Eindruck zu schaffen, müssen die Sprites so überlagert werden (z-Order Sortierung), dass bestimmte Sprites der Betrachtungsperson näher erscheinen als andere. Eine Skalierung damit hintere Gegenstände kleiner erscheinen als vordere erfolgt dabei nicht. Diese z-Order Sortierung muss dynamisch erfolgen, da der Nutzer durch Interaktion mit den Elementen die Lage im Raum (auch in der Tiefe, z. B. beim Aufnehmen eines Elementes) verändern kann. Noch komplizierter wird es bei Behältern, die andere Elemente aufnehmen können. Behälter müssen in mehrere Sprites (Vordergrund, Hintergrund und „Aufnahmebereich“) unterteilt werden, was einen zusätzlichen Aufwand bei der Erstellung der Grafiken zur Folge hat (s. Abbildung 1.2).

Kollisionserkennung

Eine weiteres Problem, das durch die fehlende Tiefeninformation entsteht, ist die Entdeckung von Kollisionen. [Tho95] spricht zwar davon, dass Sprites kollidieren können, dies gilt aber nur hinsichtlich ihrer zweidimensionalen Ausbreitung. In GENLAB wurde die Kollisionserkennung für stehende Objekte durch Hinzunahme weiterer unsichtbarer Sprites für jedes Element, die die Standfläche definieren, realisiert. Trifft ein Element mit seiner Standfläche mit der Standfläche eines anderen Elements zusammen, entspricht dies einer Kollision. Diese Form der Kollisionserkennung hängt von Wahl der Standfläche ab. Die Höheninformation von Objekten wird nicht berücksichtigt, wodurch Ungenauigkeiten bei der Erkennung der Kollisionen entstehen. In GENLAB können diese Ungenauigkeiten toleriert werden. Es sind aber Experimente beispielsweise in der Physik vorstellbar, in denen solche Ungenauigkeiten unzulänglich sind.

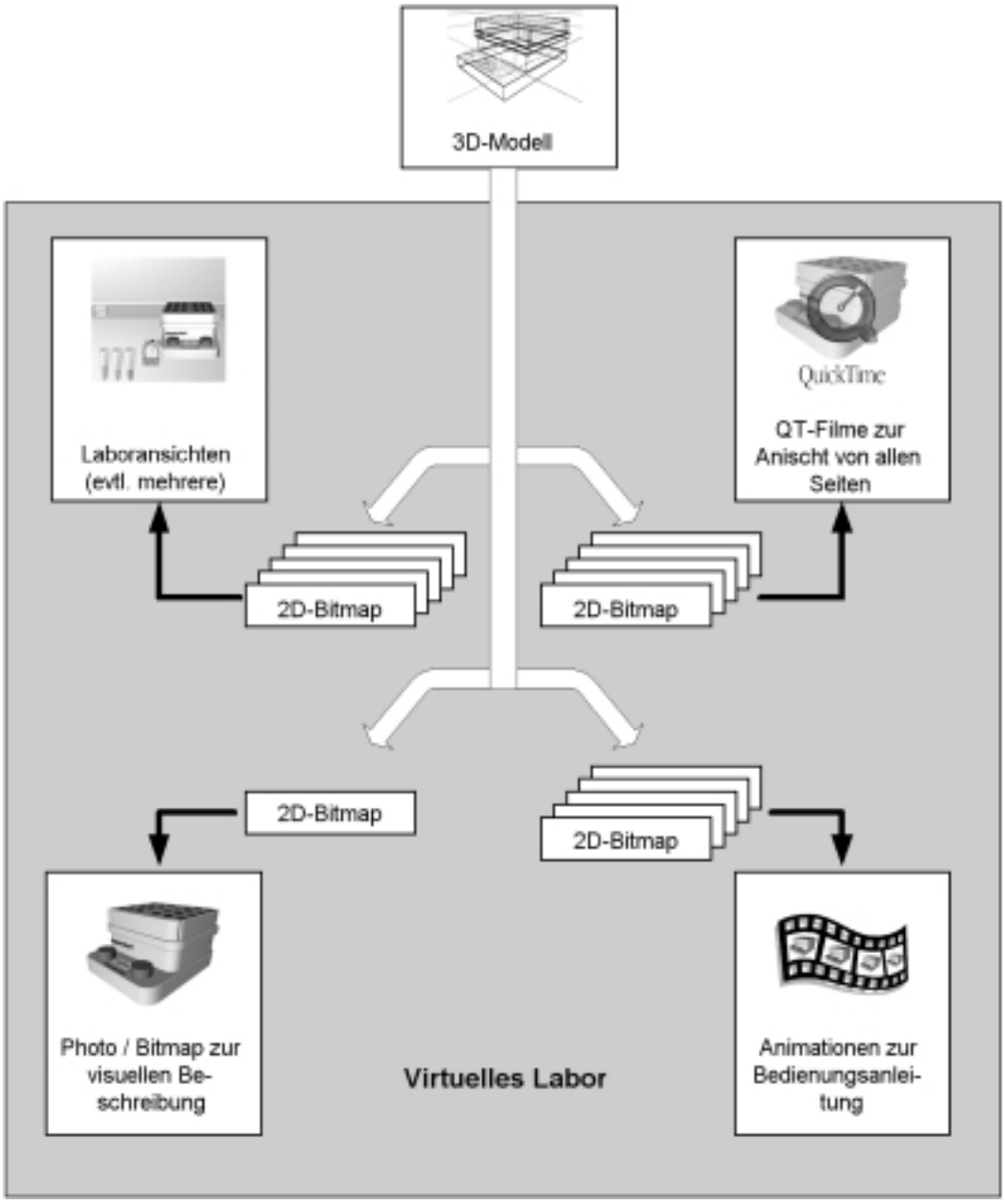

Abbildung 1.4: Verschiedene Repräsentationen im 2D-Labor setzen sich aus unterschiedlichen Quellen zusammen

Perspektive

Für bewegliche Objekte sind perspektivische Veränderungen nur schwer oder gar nicht durch Transformation der Sprites darstellbar. Eine Verschiebung eines Elementes nach hinten oder vorne kann durch Skalierung der Sprites erfolgen (in GENLAB nicht eingebaut). Auch eine Rotation um die z-Achse ist in einem bestimmten Rahmen möglich. Sollen Elemente in der horizontalen oder vertikalen Richtung verschoben werden, fällt schon eher auf, dass die Perspektive der Elemente nicht mehr korrekt ist. Drehungen um die x- oder y-Achse sind gar nicht möglich, da diese Informationen nicht in den zweidimensionalen Bitmaps gespeichert sind. Ebenso sind Bewegungen des „virtuellen“ Nutzers nicht möglich. Dies bedeutet einen erhöhten Aufwand bei der Gestaltung der Navigation zwischen Arbeitsflächen und Laborräumen.

1.1.2 3D Visualisierung

Mit Hilfe einer echten 3D-Darstellung können die im vorigen Abschnitt genannten Probleme gelöst oder vereinfacht werden. Die Idee bei einer echten 3D-Darstellung ist die direkte Integration des 3D-Modells in das virtuelle Labor. Alle Repräsentationen greifen für ihre Visualisierung auf das 3D-Modell zu. Abbildung 1.5 verdeutlicht diesen Vorgang. Dies führt zu erheblichen Vorteilen bei der Generierung und Aktua-

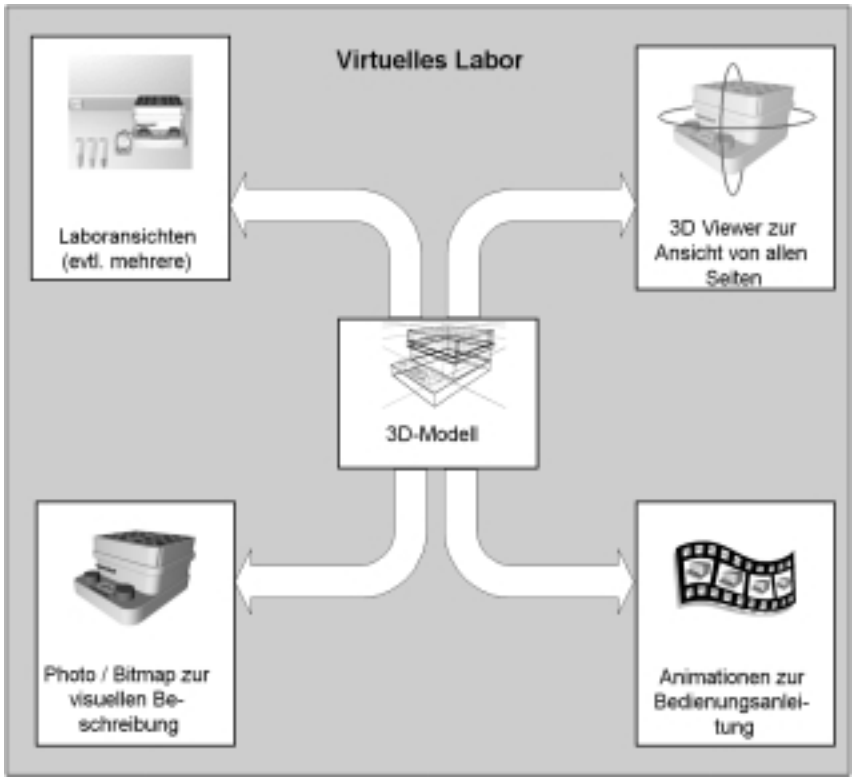

Abbildung 1.5: Im 3D-Labor greifen alle Repräsentationen auf die gleiche Quelle zurück

lisierung der Laborkomponenten. Das „Zerschneiden“ und wieder Zusammensetzen bei der Erstellung eines interaktiven Views entfällt, da jedes einzelne Bestandteil der Komponente hierarchisch mit Positions- und Orientierungsangaben in dem 3D-Modell gespeichert ist und über einen Namen referenziert werden kann. Änderungen an einer Laborkomponente müssen nur an dem 3D-Modell durchgeführt werden und werden sofort an allen Repräsentationen sichtbar. Das erhöht sowohl die Wartbarkeit als auch die Konsistenz der verschiedenen Darstellungen. Bei einer Aktualisierung über das Internet müssen nur das 3D-Modell, seine Texturen und Materialien übertragen werden. Die eigentlichen Informationen, d. h. die Pixelwerte eines Bildes werden lokal zur Laufzeit erzeugt und nicht übertragen, was zu einem wesentlich geringeren Datenvolumen führt. Da das 3D-Modell über Tiefeninformationen verfügt, wird eine Z-order Sortierung überflüssig. Eine genaue Kollisionserkennung kann aufgrund der geometrischen Struktur erfolgen. Die zur Laufzeit berechnete Darstellung der Laborkomponenten sorgen für eine immer perspektivisch korrekte Darstellung.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist die Entwicklung und Implementierung von Konzepten, die eine direkte Integration dreidimensionaler Modelle in virtuelle Labore ermöglichen. Dieses Ziel soll auf Basis des GENLAB-Frameworks unter Erarbeitung folgender Teilaufgaben erreicht werden:

① Wahl einer Entwicklungsumgebung, die für die 3D-Visualisierung und Erstellung virtueller Labore geeignet ist. Dabei müssen folgende Aspekte berücksichtigt werden:

- ☞ Fähigkeit, 3D-Modelle darzustellen und zu importieren,
- ☞ Art und Mächtigkeit der Programmiersprache,
- ☞ Medienverwaltung,
- ☞ Bedienbarkeit durch Nicht-Informatiker und Multimedia-Designer und
- ☞ die Möglichkeit der Anbindung des GENLAB-Frameworks

② Erweiterung des GENLAB-Frameworks um abstrakte Klassen, die eine Integration interaktiver dreidimensionaler Modelle in virtuelle Versuchsumgebungen ermöglichen.

③ Definition eines Workflows zur Generierung einer virtuellen Versuchsumgebung mit integrierten 3D-Geräten in Form eines Leitfadens. Dies umfasst sowohl die Konstruktion, den Import der 3D-Modelle, sowie die Programmierung notwendiger Methoden.

④ Implementierung eines prototypischen virtuellen Labors unter Verwendung des GENLAB-Frameworks und der in dieser Arbeit entworfenen Erweiterung für die 3D-Unterstützung.

1.3 Gliederung

Die vorliegende Arbeit ist in sieben Kapitel gegliedert. Im folgenden Kapitel werden allgemeine Anforderungen an virtuelle Labore sowie spezifische Anforderungen an die 3D-Visualisierung erörtert. Kapitel 3 analysiert bestehende Entwicklungsumgebungen. Die für die Ziele dieser Arbeit geeigneteste Entwicklungsumgebung wird für das weitere Vorgehen ausgewählt. Anschließend werden in Kapitel 4 Konzepte zur 3D-Visualisierung für virtuelle Labore entworfen. Die Beschreibung der technischen Umsetzung der Konzepte folgt im 5. Abschnitt. Das Kapitel 6 beinhaltet einen Entwicklerleitfaden, der den gesamten Entwicklungsprozess von der Konstruktion eines 3D-Modells bis zur Erstellung einer prototypischen Versuchsumgebung umfasst. Im letzten Kapitel werden die vorliegenden Ergebnisse zusammengefasst und bewertet. Weiterhin werden Vorschläge für zukünftige Entwicklungen und Ansatzpunkte für weitere Forschungsarbeiten gegeben.

2 Problemanalyse und Anforderungen

Im vorliegenden Kapitel werden die Anforderungen an eine dreidimensionale Visualisierung im Kontext virtueller Labore gestellt. Der folgende Abschnitt beinhaltet eine Analyse der Eigenschaften und Bestandteile virtueller Labore. Daraus ergeben sich mögliche Einsatzszenarien und Anforderungen an eine Integration dreidimensionaler Modelle in virtuelle Labore. Sie lassen sich klassifizieren in Anforderungen an die Darstellung (Abschnitt 2.2), Interaktion und Manipulation (Abschnitt 2.3) sowie Anforderungen an die Erstellung und Beschaffenheit der 3D-Modelle (Abschnitt 2.4). Im Abschnitt 2.5 werden Erfordernisse an eine Datenstruktur zum Zugriff auf die Objekte im Labor analysiert. Die Konstruktion der Modelle wird üblicherweise in professioneller 3D-Grafiksoftware vorgenommen. Deshalb muss ein gemeinsames 3D-Grafikformat verwendet werden, das die Übertragung der Modelle zwischen 3D-Software und Laborumgebung ermöglicht. Die Anforderungen an dieses Format werden in Abschnitt 2.6 festgehalten. Der letzte Abschnitt fasst die Ergebnisse dieses Kapitels zusammen.

2.1 Allgemeine Eigenschaften und Anforderungen virtueller Labore

Virtuelle Labore und Experimentierumgebungen haben unabhängig von der 3D-Visualisierung bestimmte Anforderungen und Charakteristika. Diese wurden in [Elf99], [AAH+98] und [O+97] ausführlich erörtert. Da sie z. T. jedoch für die weiteren Überlegungen bzgl. der Erweiterung um Dreidimensionalität wichtig sind, werden hier einige Aspekte nochmals aufgegriffen. Die nachfolgenden Beschreibungen stützen sich auf die o. g. Literatur und auf das Projekt GENLAB.

Nach den Untersuchungen der Projektgruppe „Virtuelle naturwissenschaftlich-technische Labore im Internet“ [AAH+98] kann ein virtuelles Labor in die zwei Teile Rahmenhandlung und Experimentierumgebung gegliedert werden. Die Rahmenhandlung vermittelt das theoretische Wissen über Versuchskomponenten (Behälter, Geräte, Substanzen etc.) und Versuchsabläufe (Anleitungen, Animationen). Sie kann durch folgende Elemente charakterisiert werden:

- **Text und statische Bilder:** Ein Großteil des theoretischen Wissens kann durch Text und statischen Bildern vermittelt werden. Diese Medien reichen aus, um einfache Sachverhalte klar darzustellen.

- **schematische Animationen:** Hintergrundwissen und interne Versuchsvorgänge (z. B. chemische Reaktionen, Molekühlbewegungen) können durch schematische Animationen verdeutlicht werden. Mittels geringer Eingriffsmöglichkeiten des Lernenden in die Animationen beispielsweise durch Veränderung einiger Parameter kann bestimmtes Wissen „spielend“ vertieft werden.

- **Videos:** Videos und Quicktime-VR Filme können genutzt werden, um komplexe Vorgänge oder Gegenstände visuell und akustisch realitätsnah zu vermitteln. Die Interaktion des Nutzers ist dabei gering.

- **Hyperlinks:** Die Verwendung von Hyperlinks zur Navigation innerhalb komplexer Informationsstrukturen ist ein Schlüsselelement der modernen Wissensvermittlung und erleichtert den Zugang zu assoziativen Informationen eines Themenbereichs.

- **Unterstützung unterschiedlicher Lernformen:** Die verschiedenen Medien werden nicht ausschließlich verwendet, sondern bieten erst in Kombination eine Unterstützung unterschiedlicher individueller Lernformen.

Neben der Rahmenhandlung stellt die Experimentierumgebung den zweiten Teil eines virtuellen Labors dar. Sie dient zur Vermittlung praktischen Wissens. Folgende Charakteristika zeichnen die Experimentierumgebung aus:

- **realitätsnahe Darstellung und Interaktion:** Die Übertragung der in der virtuellen Umgebung erlernten Tätigkeiten kann nur dann schnell und leicht vollzogen werden, wenn die virtuelle Welt möglichst realitätsnah dargestellt wird. Die virtuellen Laborkomponenten müssen deshalb ein ihrer realen Vorbilder ähnliches Aussehen und Bedienmöglichkeiten vorweisen.

- **komplexe Versuchsabläufe und ihre Steuerung:** Komplexe Versuchsabläufe werden durch einen erheblichen Programmieraufwand detailliert modelliert und präsentiert. Dazu gehören nicht nur räumliche, sondern auch zeitliche Abhängigkeiten.

- **Protokollierung der Versuchsergebnisse:** Durchgeführte Arbeitsschritte werden vom System erkannt und mitprotokolliert.

- **Unterstützung des Lernenden:** Dem Lernenden wird Unterstützung in der Durchführung der Versuche geboten – zusätzlich zu dem theoretischen Wissen aus der Rahmenhandlung. Dazu gehören beispielsweise Orientierungshilfen bzgl. des Versuchkontextes (Anzeige der bisher erledigten und durchführbaren Arbeitsschritte), Meldungen der vom System erkannten Fehler, Undo-Funktionalität und Animations- oder Vorführmöglichkeiten des Versuchablaufens.

☞ **Navigation zwischen Laborräumen:** Der Anwender kann sich in der virtuellen Welt innerhalb und zwischen Laborräumen bewegen.

Aus der Sicht der Erstellung eines virtuellen Labors kann die Rahmenhandlung zusammenfassend als netzwerkartiges, größtenteils statisches Informationsangebot mit geringen interaktiven Elementen beschrieben werden. Der Einsatz hypermedialer und multimedialer Konzepte ist maßgeblich. Bei Gebrauch geeigneter Werkzeuge kann nahezu auf Programmierung verzichtet werden. Im Gegensatz dazu stellt sich die Experimentierumgebung als komplexe, interaktionsreiche und dynamische Komponente dar. Die komplizierten Vorgänge, Abhängigkeiten und Interaktionsangebote können nur durch hohen Programmieraufwand realisiert werden.

Aus diesen vielfältigen Eigenschaften virtueller Labore ergeben sich spezifische Anforderungen an die Entwicklung. Die Integration von Fachwissen, Didaktik, Grafik- und Multimedia-Design und Programmierung erfordert zudem ein Projektteam aus Personen mit verschiedenen Qualifikationen und Fähigkeiten. Dennoch sollen die von den Entwicklern erstellten Komponenten nahtlos in ein Produkt integriert werden.

Sowohl die Rahmenkomponente als auch die Experimentierumgebung zeigen oft verschiedene Repräsentationen desselben Objektes. Sie unterscheiden sich in der Darstellung und in den Interaktionsmöglichkeiten. Während im zweidimensionalen Labor für jede Repräsentation eine andere Informationsquelle genutzt wird, kann durch die Integration dreidimensionaler Modelle auf dieselbe Informationsquelle für alle Instanzen eines Objektes zugegriffen werden (siehe auch Abbildungen 1.4 und 1.5). Daraus ergeben sich Anforderungen an die Interaktions- und Manipulationsmöglichkeiten im virtuellen Labor. Dazu gehören die Möglichkeit interaktiv und direkt Geräte zu bedienen, z. B. Knöpfe und Schalter zu betätigen, Laborkomponenten zu bewegen und ineinander zu stellen sowie schnell zu Informationen über die Komponente zu gelangen. Die daraus ableitbaren Anforderungen an die 3D-Modelle und ihre Darstellung werden in den beiden folgenden Abschnitten genau betrachtet.

2.2 Berechnung und Darstellung der 3D-Szene

Die Darstellung einer realistischen 3D-Szene wird in der Regel durch zwei Faktoren bestimmt:

① Qualität der Darstellung

② Geschwindigkeit der Darstellung

In einem virtuellen Labor sind beide Faktoren wichtig. Grundvoraussetzungen sind eine möglichst realistische *und* aufgrund der Interaktionen in der Experimentierumgebung eine schnelle Darstellung auf üblichen Multimedia-Rechnern. Von einem Modell einer Szene bis zur realistischen Darstellung am Monitor sind mehrere teilweise gleichzeitig, teilweise hintereinander durchführbare Berechnungsschritte notwendig. Für die

einzelnen Berechnungen gibt es verschiedene Verfahren, welche die o. g. Faktoren unterschiedlich beeinflussen. [FD94] gibt u. a. folgende Berechnungsschritte für eine realistische Darstellung an:

- Algorithmus zur Ermittlung sichtbarer Flächen
- Beleuchtungsmodell
- Schattierungsmodell
- Reflexion
- Transparenz
- Schatten
- Berücksichtigung von Materialeigenschaften / Texturen
- Projektion (Abbildung der 3D-Darstellung auf eine 2D-Ebene)

Diese Schritte und einige ihrer Berechnungsverfahren werden im Folgenden kurz beschrieben und auf Relevanz für eine 3D-Visualisierung für virtuelle Labore geprüft.

2.2.1 Ermittlung sichtbarer Flächen

Das Ziel bei einer dreidimensionalen Darstellung ist es, nur die vom Betrachter sichtbaren Flächen zu berechnen. Nicht sichtbare Flächen können zur Verringerung der Berechnungszeit entfernt werden. Es gibt verschiedene Verfahren zur Ermittlung sichtbarer Flächen. Einige werden in [FD94] vorgestellt und genau beschrieben. Darunter sind die Verfahren:

- Depth-sort,
- z-Buffer,
- Scanline und
- Raytracing.

Unter anderem kann die Berechnungsdauer von der Anzahl der Polygone in einer Szene ab abhängen. Die Tabelle aus [FD94] verdeutlicht diese Aussage, indem die verschiedenen Algorithmen auf Szenen mit unterschiedlicher Polygonzahl angewendet werden und ihre relative Performance gemessen wird:

Algorithmus	**100 Polygone**	**2500 Polygone**	**60000 Polygone**
Depth-sort	1[a]	10	507
z-Buffer	54	54	54
Scan-Line	11	64	307

[a] Die Einträge sind bezüglich dieses Falls normalisiert

Das Raytracing-Verfahren wurde in der Tabelle nicht mit aufgenommen, da es in seinem Algorithmus gleichzeitig die Schattierung mit integriert und deswegen nicht mit den anderen Verfahren direkt vergleichbar ist. Für eine dynamische Berechnung ist es ohnehin wegen der hohen Rechenintensitivität nicht geeignet.

Eine komplexe Szene in GENLAB besteht aus mehreren zehntausend Polygonen[1]. Oft sind der z-Buffer und Scan-Line Algorithmus in 3D-Grafikkarten implementiert und unterscheiden sich nur noch unwesentlich. Beide Verfahren sind daher zur Ermittlung sichtbarer Flächen in virtuellen Laboren geeignet.

2.2.2 Beleuchtung

Die in der Realität vorherrschenden Lichtquellen lassen sich in die Kategorien Punktlicht, Spotlicht, Umgebungslicht und Distanzlicht einteilen. Jede Lichtquelle besitzt Eigenschaften wie Farbe, Intensität, Abfall und atmosphärische Abschwächung. In virtuellen Welten versucht man diese Lichtquellen nachzubilden und nennt sie *Beleuchtungsmodelle*. Die Berechnungsdauer der Schattierung für einen Pixel kann rapide mit der Anzahl der Lichtquellen steigen. Neben dem Umgebungslicht, das die gesamte Szene gleichermaßen erhellt, sollten zumindest Punktlichtquellen eingesetzt werden können, um die Laborumgebungen realistisch darzustellen.

2.2.3 Schattierung

Der Begriff *Schattierungsmodell* wird oft im Zusammenhang mit dem Begriff *Beleuchtungsmodell* verwendet. Genauer legt das Schattierungsmodell fest, wann und wie ein Beleuchtungsmodell angewandt wird. Es gibt viele Darstellungsalgorithmen (Schattierungsmodelle), die Performance und Aussehen bestimmen. Darunter fallen die einfachen aber unrealistischen Verfahren wie Skelett- und Drahtgittermodelle, sowie die Schattierungsmethoden mit mittlerer Qualität Flat-, Phong- und Gouraudshading sowie die komplexen und sehr realistischen Verfahren Raytracing und Radiosity (s. [FD94]). Die Phong-Schattierung liefert die besseren Ergebnisse bei der Darstellung von Glanzlichtern gegenüber der Flat- und Gouraud-Schattierung, verursacht jedoch bei komplexeren Szenen einen nicht mehr tragbaren Rechenaufwand in interaktiven Laboren. Die Verfahren Raytracing und Radiosity zählen zu den *globalen* Schattierungsalgorithmen. Sie berechnen die Farbe eines Pixels nicht nur aus den Lichtquellen und des Objektmaterials, sondern beziehen Reflexionen und Brechungen anderer Objekte mit ein. Sie eignen sich nicht für die Verwendung in interaktiven Umgebungen. Eine Anforderung an die Darstellung der Laborkomponenten in virtuellen Laboren ist möglichst realistisch und auf üblichen Multimedia-PCs performant zu sein. Einen Kompromiss zwischen diesen beiden Eigenschaften beschreiten die Flat- und Gouraud-Schattierungs Verfahren.

[1] Die Polygonzahl wurde in GENLAB nicht optimiert, da die Modelle nicht in die Entwicklungsumgebung importiert wurden, sondern nur vorher berechnete Bitmaps.

Bei allen vorgestellten Verfahren hängt die absolute Berechnungsdauer u. a. von der Anzahl der sichtbaren Polygone ab. In einigen Situationen müssen wesentlich mehr Polygone schattiert werden als in anderen. Bei einer Navigation durch virtuelle Räume werden mehr Geräte sichtbar als bei der Betrachtung einer Arbeitsfläche. Wünschenswert sind Algorithmen, die sich der Situation anpassen können und jeweils ein geeignetes Schattierungsmodell wählen, die Polygonzahl weit entfernter Objekte verringern (*Depth Cueing*), Nebel einstreuen (*Fogging*) oder eine Unschärfe simulieren. Diese Verfahren sind nicht notwendig zur 3D-Visualisierung, sondern verbessern lediglich die Performance der Darstellung in manchen Situationen.

2.2.4 Reflexionseigenschaften

Die Berechnung echter Reflexion (Spiegelung) in interaktiven realitätsnahen Umgebungen auf Standard-Rechnern ist in angemessener Zeit noch nicht möglich - dazu wären die komplexen Schattierungsverfahren Raytracing oder Radiosity notwendig. Echte Reflexionen stellen daher keine Anforderung an die 3D-Darstellung im Labor da.

2.2.5 Transparenz

In realen naturwissenschaftlichen Laboren findet man viele transparente Gegenstände vor. Transparenz muss deswegen im virtuellen Labor angemessen dargestellt werden können. Sie entsteht dadurch, dass Lichtstrahlen nicht auf der Oberfläche des Objektes reflektieren, sondern ganz oder zum Teil hindurchgelassen werden. In der Realität werden diese Strahlen im Normalfall gebrochen, d. h. sie erfahren eine Ablenkung um einen bestimmten Wert, der durch die Dichte des Materials und der Tiefe des Objektes berechnet wird. Ähnlich wie Reflexionen kann Transparenz mit Brechung nur in aufwendigen Berechnungen wie Raytracing berücksichtigt werden. Ohne Brechung lässt sich Transparenz auch in den einfacheren Algorithmen wie z-Buffer und Scan-Line integrieren und ist deshalb in der 3D-Visualisierung zu realisieren.

2.2.6 Schatten

Schatten entstehen, wenn Lichtstrahlen von einem nichttransparenten Objekt aufgehalten werden und eine dahinterliegende Fläche nicht erreichen können. Um „echten“ dynamischen Schatten zu berechnen, müssten alle Lichtstrahlen von allen Beleuchtungsquellen verfolgt und auf Schnitt mit einem Objekt überprüft werden. Dieses Vorgehen kann in einem Scan-Line und Raytracing-Verfahren integriert werden. Der Rechenaufwand für Scan-Line-Verfahren steigt dabei stark an. Schatten verstärken den Eindruck einer realen Umgebung und können unter Umständen bei der Bewegung von Laborkomponenten eine Orientierungshilfe sein, sind aber nicht unbedingt für die dreidimensionale Darstellung der Objekte erforderlich.

2.2.7 Berücksichtigung von Materialeigenschaften / Texturen

Die Materialeigenschaften eines Objektes bestimmen in der 3D-Darstellung die Ablenkung der Lichtstrahlen, die auf das Objekt treffen. Gläserne Objekte lenken die Strahlen nur wenig ab. Wasser lässt sie mit einer gewissen Ablenkung hindurch. Manche Objekte reflektieren die Strahlen entsprechend ihrem Einfallswinkel. Wie oben gezeigt kann nicht jeder Algorithmus alle Materialeigenschaften darstellen. Ersatzweise können auch Texturen verwendet werden, um für ein Material charakteristische Eigenschaften zu simulieren. So können nicht nur die Farbe des Objektes beeinflusst werden, sondern auch Unebenheiten auf einer Oberfläche, Reflektionen und Transparenz.

Während die Berücksichtigung der Materialeigenschaften nur rein visuelle Vorteile für den Anwender hat, kann eine Texturierung auch die Modellierung eines 3D-Objektes vereinfachen. Sie kann insbesondere zur Darstellung von digitalen Anzeigen, wie sie bei Mikrowellen oder Waagen vorkommen, benutzt werden, anstelle jeden einzelnen Bestandteil (jede LED) zu modellieren. Um Texturen auf Objekte platzieren zu können, gibt es u. a. folgende Projektionsarten: Planar Mapping, Parametric Mapping, Cylindrical Mapping, Spherical Mapping und UV Mapping. Die letzte Methode berücksichtigt als einzige die Polygonstruktur der Objekte. Die anderen Methoden können sich beliebigen Objekten nur bedingt anpassen. Da es sich in virtuellen Laboren um beliebig geformte Laborkomponenten handelt, ist das UV-Mapping eine wünschenswerte Berechnungsmethode bei der Darstellung der Objekte.

2.2.8 Dreidimensionale Darstellung / Projektion

Ein Problem der dreidimensionalen Darstellung entsteht durch die Abbildung dreidimensionaler Modelle auf ein zweidimensionales Ausgabegerät (Monitor). Als hilfreiche Metapher bei der Erzeugung dreidimensionaler Szenen hat sich die *synthetische Kamera* erwiesen. Die Kamera wird unter anderem durch Position und Richtung definiert. Beim Betätigen des Auslösers wird ein zweidimensionales Abbild der Szene erzeugt. In interaktiven Umgebungen muss der Auslöser der Kamera bei jeder Änderung der Szene gedrückt und die dreidimensionale Szene auf den 2D-Bildschirm *projiziert* werden. Dabei werden jeweils nur die Veränderungen neu berechnet und dargestellt. Bei einfachen Objektmanipulationen ist die Anzahl der Veränderung gering. Eine Bewegung des Betrachters durch den virtuellen Raum bedeutet allerdings eine kontinuierliche Änderung der Kameraposition und hat eine vollständige Berechnung und Darstellung der gesamten 3D-Szene zur Folge. Es gibt mehrere Projektionarten, die je nach Anwendungsdomäne unterschiedlich geeignet sind. Im Themenbereich dieser Arbeit sind nur *planare geometrische Projektionen* interessant, welche die Szene auf eine Ebene und nicht auf gekrümmte Flächen abbilden. Nach [FD94] lassen sich die Projektionsarten lassen in zwei Kategorien einteilen, die *Zentralprojektion* und *Parallelprojektion*. Bei einer *Zentralprojektion* haben das Projektionszentrum und die Projektionsebene endlichen Abstand zueinander. Die Projektoren schneiden

sich. Das Projektionszentrum bei einer *Parallelprojektion* befindet sich unendlich weit weg von der Projektionsebene, was zu parallelen Projektoren führt (s. Abbildung 2.1). Eine Zentralprojektion ähnelt der Projektion des menschlichen Auges. Sie hat den Ef-

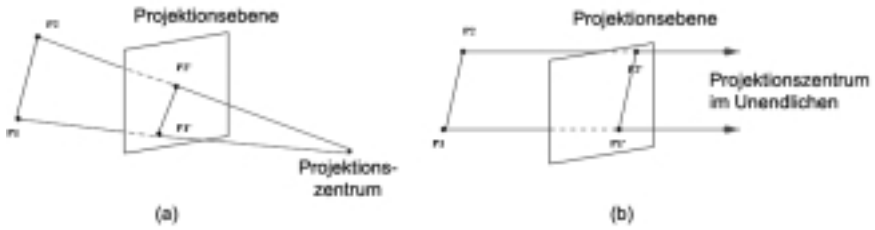

Abbildung 2.1: Zwei verschiedene Projektionen der selben Linie. (a) Zentralprojektion, (b) Parallelprojektion.

fekt der *perspektivischen Verkürzung*. Die Größe der Zentralprojektion eines Objekts verhält sich umgekehrt zum Abstand des Objekts vom Projektionszentrum. Die Parallelprojektion liefert weniger realistische Darstellungen, da die perspektivische Verkürzung und dadurch die für virtuelle Labore wichtige Vermittlung einer räumlichen Tiefe fehlt. Ein Anforderung für die Darstellung der 3D-Umgebung ist daher die Verwendung einer zentralen Projektion.

Zusätzlich zu den von [FD94] genannten Faktoren gibt es weitere Einflüsse auf die Darstellung einer Szene:

2.2.9 Hardwareunterstützung

Einige der üblichen multimediafähigen Rechner werden mittlerweile mit einer 3D-Grafikkarte ausgeliefert. Diese Grafikkarten verarbeiten mit einem eigenen Prozessor und oft zusätzlichem Speicher „High-Level“ 3D-Befehle, wie sie in OPENGL oder DIRECT3D vorkommen. Dies entlastet den Hauptprozessor und ist wegen der Berechnung auf Hardware-Ebene ein vielfaches schneller als durch Software. Da keine statistischen Untersuchungen gefunden wurde, die belegen können, dass 3D-Grafikkarten in den meisten Multimedia-Rechnern vorhanden sind, muss die 3D-Darstellung auch ohne 3D-Hardwareunterstüzung funktionsfähig sein.

2.2.10 Zusammenfassung

An die Berechnung und Darstellung der 3D-Szene werden zusammenfassend folgende Anforderungen gestellt, damit eine geeignete Darstellungsqualität und -geschwindigkeit erreicht werden kann:

- ☞ Ermittlung der sichtbaren Flächen durch z-Buffer oder Scan-Line Verfahren
- ☞ Integration von Umgebungs- und Punktlichtquellen

- ☞ Flat- oder Gouraud-Schattierung
- ☞ einfache Transparenz ohne Brechung
- ☞ Texture Mapping
- ☞ 3D-Darstellung durch Zentralprojektion
- ☞ Berechnung und Darstellung der 3D-Modelle auf der Basis von Software

2.3 Interaktion und Manipulation der 3D-Objekte

Der Interaktionsgrad mit den Laborkomponenten in der Experimentierumgebung des Labors ist äußerst hoch. Der Anwender muss in der Lage sein, Geräte zu bedienen und miteinander interagieren zu lassen. Die Möglichkeiten zur Interaktion sind in einer dreidimensionalen Umgebung vielfältiger als in einer zweidimensionalen. Neben den Manipulationen eines Objektes und der Interaktion mehrerer Objekte miteinander muss untersucht werden, welche Anforderungen an die Navigation und Orientierung in der 3D-Szene gestellt werden müssen. Nach [FD94] gibt es folgende grundlegende 2D und 3D-Interaktionsaufgaben (*basic interaction tasks, BITs*):

- ☞ Positionierung
- ☞ Rotation
- ☞ Auswahl
- ☞ Texteingabe
- ☞ Eingabe numerischer Werte

Die letzten beiden Elemente werden nicht weiter berücksichtigt, da sie keinen Einfluss auf eine 3D-Visualisierung nehmen. Die Positionierung, Rotation und Auswahl gibt es ebenfalls in zweidimensionalen Umgebungen, sind aber in 3D-Applikationen komplizierter. Insbesondere nennt [FD94] zwei Probleme bei den 3D-Interaktionen:

① schwierige Wahrnehmung der dreidimensionalen Tiefenverhältnisse zwischen einem Cursor und einem Objekt

② Bewegungen der 2D-Interaktionsgeräte müssen in die dritte Dimension abgebildet werden

[FD94] gibt ebenfalls Ideen zur Lösung dieser Probleme. Sie weichen auf die Darstellung mehrerer 2D-Ansichten aus und zwingen den Anwender zu mühsamen Aktionen. Sie eignen sich deswegen nicht in virtuellen Laboren. Inwiefern die o. g. Interaktionsaufgaben für dreidimensionale virtuelle Labore relevant sind, zeigt die folgende Aufstellung:

2.3.1 Positionierung

Bewegliche Objekte müssen in alle Richtungen, also so wohl in der horizontalen und vertikalen Richtung wie auch in die Tiefe, verschiebbar sein. Einige Elemente können dabei Restriktionen unterliegen. Schieberegler dürfen beispielsweise nur in eine Richtung und über eine bestimmte Distanz verschoben werden. Bei einer Positionierung müssen sowohl die Perspektive und die Größe des Objektes durch das System angepasst werden.

2.3.2 Rotation

Alle beweglichen Laborkomponenten sollen rotierbar sein. Die Rotationsachsen können je nach Objekt gesperrt oder freigegeben werden. Rotationen können Einschränkungen unterliegen. Beispielsweise können Türen oder einige Schalter, nur bis zu einem bestimmten Winkel um eine festgelegte Achse gedreht werden.

2.3.3 Auswahl

Dem Anwender muss eine Selektion eines Objektes mit allen seinen Unterobjekten ermöglicht werden. Ebenso muss eine Selektion eines Unterobjektes ohne Auswahl dessen Oberobjektes realisierbar sein.

Die folgenden Abschnitte stellen nicht direkt eine Interaktionsaufgabe dar, sondern erläutern zusätzliche Anforderungen an Interaktionen in virtuellen Laboren:

2.3.4 Kollisionsentdeckung

Bei der Bewegung eines Objekts können Kollisionen mit anderen Objekten stattfinden. Diese Art der Interaktion zwischen zwei Objekten ist ein elementarer Bestandteil innerhalb der Experimentierumgebung. Auf diese Weise können beispielsweise Behälter direkt interaktiv befüllt oder Objekte in andere Geräte gestellt werden. Kollisionen müssen von der Software während einer Positionierung erkannt werden. Man spricht dabei von *Kollisionsentdeckung* oder *Kollisionserkennung*. Hierfür gibt es mehrere Verfahren. Das einfachste ist das *Bounding-Box*-Verfahren. Hier bildet nicht die polygonale Struktur der Objekte die Grundlage für Überschneidungen, sondern der Quader, der gerade so groß ist, dass er das Objekt vollständig umgibt. Bei der Untersuchung auf Kollisionen werden nur Überschneidungen der Quader überprüft. Diese Art der Kollisionserkennung ist schnell in der Berechnung und einfach zu implementieren. Bei bestimmten Objekten kann es jedoch zu großen Ungenauigkeiten kommen wie Abbildung 2.2 zeigt. Hier überschneiden sich die Bounding Boxes, es entsteht ein Schnittquader, die beiden Objekte sind aber noch weit von einander entfernt.

In virtuellen Laboren sind Versuche denkbar, in denen diese Art der Kollisionserkennung nicht ausreicht. Wesentlich komplexere Algorithmen analysieren die genauen

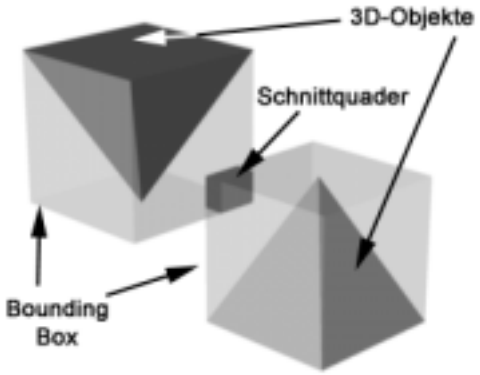

Abbildung 2.2: Ungenauigkeiten bei der Kollisionsentdeckung mit Bounding Boxes

Polygonstrukturen und weisen dadurch nur geringe Ungenauigkeiten auf. Einige Verfahren überprüfen nicht nur auf Kollisionen, sondern ermöglichen eine genaue Bestimmung der Kollisionsbereiche der entsprechenden Objekte. Man spricht dann von *Contact Determination*. In der Universität von North Carolina wurden mehrere für verschiedene Systeme geeignete Kollisionsentdeckungs-Verfahren entwickelt und implementiert [UNC99]. V-COLLIDE beispielsweise berechnet die konvexe Hülle eines Objektes vor und speichert sie in einer hierarchischen Repräsentation von ausgerichteten Bounding-Boxes. Diese Repräsentation stellt eine Annäherung an die polygonale Struktur des Objektes dar. Abbildung 2.3 verdeutlicht das Verfahren in 2D. Während

Abbildung 2.3: Annäherung einer linearen Struktur durch ausgerichtete Bounding-Boxes. Aus [LMCG].

der Laufzeit werden diese Bounding-Boxes auf Schnittbereiche mit anderen Bounding-Boxes überprüft. Die in Abbildung 2.4 dargestellten Stellungen der Objekte können einen Schnitt zweier Bounding-Boxes verursachen. Abb. 2.4 (a) stellt dabei eine einfache Kollision dar, die auch mit einfachen *Bounding-Box*-Verfahren hätte entdeckt werden können. In Abb. 2.4 (b) schneiden sich zwar die Bounding-Boxes, bei der Fortführung des Algorithmus würde jedoch erkannt werden, dass sich die Objekte selbst nicht schneiden. Der Fall in Abb. 2.4 (c) verhält sich ähnlich. Hierbei kommt es jedoch zu einer tatsächlichen Kollision. Der Algorithmus liefert die Kollisionsbereiche zurück. Performance-Tests wurden nur auf einer SGI durchgeführt. Die Geschwindigkeit des Algorithmus kann durch die Anzahl der Bounding-Boxes gesteuert werden. Wählt man eine geringere Approximation des Objekts, werden weniger Bounding-Boxes und damit Überschneidungsüberprüfungen benötigt. So ist die Kollisionsent-

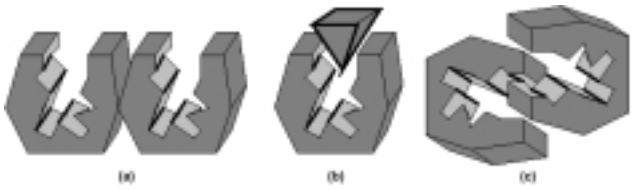

Abbildung 2.4: V-COLLIDE erkennt die Situationen richtig: (a) und (c) sind Kollisionen, (b) ist keine. Aus [LMCG]

deckung in Qualität und Geschwindigkeit an den jeweiligen Rechner und die jeweilige Aufgabe anpassbar.

2.3.5 Navigation

Ein Labor ist üblicherweise in mehrere Arbeitsbereiche und Räume aufgeteilt. Innerhalb des Labors besteht daher die Notwendigkeit der Bewegung zwischen diesen Arbeitsplätzen und Räumen, also der Navigation. Es gibt zwei verschieden Möglichkeiten des Navigierens: die kontinuierliche und die diskrete Navigation. Beim kontinuierlichen Navigieren kann der Anwender die Kamera rotieren und durch einen Raum bewegen. Ein Herumandern durch das Labor erweckt die Neugierde des Anwenders und steigert seine Motivation. Diese Vorgehensweise stellt allerdings gewisse Ansprüche an die Interaktion (wie bewegt der Anwender die Kamera?) und an die Performance. Eine Kamerafahrt hat eine kontinuierliche Neuberechnung und Darstellung der 3D-Szene zur Folge. Maßnahmen zur Steigerung der Perfomance müssen getroffen werden.

Eine diskrete Navigation kann durch Wechseln festgelegter Kamerapositionen erfolgen. Der Motivationseffekt ist zwar geringer als bei einer Kamerafahrt, jedoch sind die Anforderungen an die Interaktion geringer. Die Kamerapositionen können beispielsweise durch einfache Mausklicks verändert werden. Die Berechnung der Szene erfolgt nur bei einem Wechsel der Kamera. Zur Auswahl der Kamerapositionen können Gebäudekarten, Laboraufsichten oder Hotspots in der Laborumgebung verwendet werden.

Idealerweise können beide Navigationsarten in ein virtuelles Labor integriert werden.

2.3.6 Benutzer-System Schnittstelle

Alle o. g. Interaktionen mit den Laborkomponenten und die Navigation müssen durch den Anwender durchgeführt werden können. Die Akzeptanz einer Software bezüglich der Benutzer-System Schnittstelle hängt nach [BKH97] von folgenden Qualitätsfaktoren ab:

- Erlernbarkeit

- Bedienbarkeit
- Anwesenheit
- Informationsbeschaffung
- Geschwindigkeit
- Genauigkeit

Diese Kriterien hängen teilweise unmittelbar voneinander ab. Beispielsweise kann die Geschwindigkeit einer Interaktionsdurchführung verringert werden, wenn die Genauigkeit erhöht wird. Unter Umständen leidet die Erlernbarkeit unter der Anwesenheit zu vieler Interaktionsformen. Die Bewertung dieser Kriterien hängt von der Situation ab und kann nicht deswegen hier nicht entschieden werden.

Die für zur Bedienung des Labors verwendeten Eingabegeräte beschränken sich auf 2D-Eingabegeräte, also 2D-Maus und Tastatur. Welche dieser Geräte für eine bestimmte Interaktion gewählt wird, muss dem Entwickler des virtuellen Labors überlassen werden. Es muss lediglich sichergestellt werden, dass Interaktionen über entweder Tastatur oder Maus, sowie durch eine Kombination beider Eingabegeräte durchgeführt werden können.

Die Interaktion im dreidimensionalen Labor läßt mehrere Möglichkeiten zu als die im zweidimensionalen Labor. Damit die Bedienbarkeit des Systems nicht unter der 3D-Visualisierung leidet, sind Hilfestellungen bzgl. der Interaktionsformen anzubieten. Sie lassen sich folgendermaßen Klassifizieren:

- Hilfen zur Translation und Rotation, z. B.:
  - Einschränken der Achsen
  - Ausrichten von Objekten an einem anderen Objekt
  - gleichmäßig Verteilen
  - Einschränkungen in der Schritt-/Winkelgröße
  - Snapping
  - akustische und visuelle Hervorhebung von Kollisionen
- Hilfen zur Selektion, z. B.
  - Selektion mehrer Objekte gleichzeitig
  - Highlighting der Objekte, die durch ein Selektionskommando ausgewählt werden würden
  - Auswahl über eine Texteingabe des Objektnamens
  - Auswahl über eine andere Darstellung, z. B. einer hierarchischen Präsentation
- Hilfen zur Navigation und Orientierung (siehe oben)

- ☞ Darstellung der aktuell gewählten Interaktionsart / des Modus z. B. durch
  - ➢ Cursorwechsel
  - ➢ Icons

Welche Hilfestellungen im konkreten Fall realisiert werden, bleibt dem Entwickler überlassen und ist nicht Thema der konkreten 3D-Visualisierung.

2.4 Erstellung der 3D-Modelle

Grundlegende Anforderungen an virtuelle Labore sind realitätsnahe Bedienmöglichkeiten und eine schnelle Darstellung der 3D-Szene. Beide Faktoren hängen u. a. von den 3D-Modellen ab. Der Abschnitt 2.2 zeigt, dass die Anzahl der Polygone einer Szene die Geschwindigkeit der Schattierung und, je nach Wahl des Algorithmus, der Ermittlung der sichtbaren Flächen beeinflusst. Eine realitätsnahe Interaktionsmöglichkeit hängt wiederum mit der Genauigkeit und Detailtreue der 3D-Modelle zusammen, welche die Anzahl der Polygone bestimmt. Der folgende Abschnitt erörtert verschiedene Modellierungstechniken und deren Auswirkungen auf die 3D-Darstellung und Interaktion. Darauf folgt eine Erläuterung wie ein Objekt beschaffen sein sollte, damit eine möglichst realistische und genaue Darstellung möglich ist.

2.4.1 Modellierungstechniken

Zur Erstellung von Festkörpern gibt es zwei grundlegende Modellierungstechniken. Die Modellierung über

- ☞ Polygone / Polygonnetze (Flächenkörper) oder
- ☞ Splines.

Flächenkörper werden durch eine Menge von Polygonen gebildet. Die geometrischen Informationen der Polygone werden in einer Datenstruktur gespeichert. Eine andere Möglichkeit bietet die Modellierung über Splines und mathematischen Funktionen. Es werden keine geometrischen Informationen über die Modelle gespeichert. Lediglich eine mathematische Beschreibung dient der späteren Darstellung. Die Integration dieser Modelle in virtuellen interaktiven Umgebungen scheidet aus zwei Gründen aus. Erstens muss die Geometrie (Form und Größe) des Objektes innerhalb des Systems bei jeder Darstellung berechnet werden und verursacht daher einen erheblichen zusätzlichen Zeitaufwand. Zweitens gibt es noch keine leistungsfähigen und auf üblichen Rechnern lauffähigen Algorithmen, die in akzeptabler Zeit Kollisionen entdecken können. Die meisten 3D-Grafikprogramme bieten eine Funktion, über Splines modellierte Körper in polygonale Flächenkörper umzuwandeln.

2.4.2 Beschaffenheit der Modelle

Je genauer ein Modell seinem realen Vorbild entsprechen soll, desto mehr Polygone werden benötigt. Diese Aussage trifft insbesondere auf Objekte mit vielen Rundungen zu. Die Approximierung eines Polygongitters an eine kurvige Fläche wird durch Verkleinerung der Polygongröße vollzogen. Ab einem bestimmten Grad der Annäherung hat die weitere Unterteilung der Polygone keine Wirkung mehr auf die Darstellungsqualität. Abbildung 2.5 veranschaulicht dies.

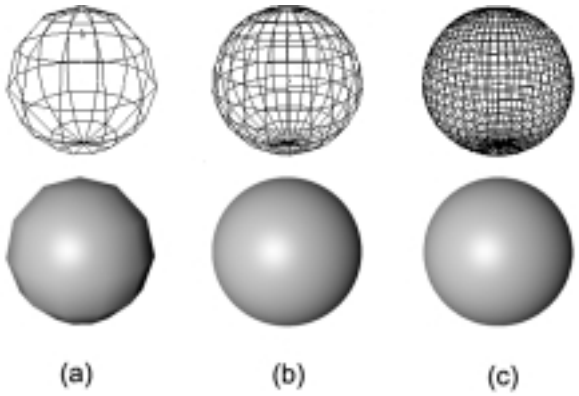

Abbildung 2.5: Aus Polygonen zusammengesetzte Kugeln: (a) = 72 Polygone, (b) = 288 Polygone und (c) = 1152 Polygone

Eine mit nur 72 Polygonen dargestellt Kugel weist in der Darstellung noch deutliche Kanten auf. Die Erhöhung auf 288 Polygonen bewirkt eine optimale Darstellung, die auch nicht durch die Approximation durch 1152 Polygone verbessert werden kann. Eine Anforderung an die Beschaffenheit eines 3D-Modells für die Integration in virtuelle Labore ist eine Polygonanzahl, die sowohl eine realistische wie auch eine schnelle Darstellung ermöglicht.

Unabhängig von der geometrischen Qualität muss ein 3D-Modell so konstruiert werden, dass alle für das Objekt vorgesehenen Interaktionen durchführbar sind. Im Arbeitsablauf zur Entwicklung auf zweidimensionaler Bitmaps basierenden Objekte werden die für Interaktionen benötigten Elemente auseinander „geschnitten“ (s. Abb. 1.2). So sind sie getrennt manipulierbar. Diese Trennung muss bei der Erstellung dreidimensionaler Views schon bei der Konstruktion des Modells erfolgen. Beispielsweise müssen Schalter eines Gerätes als eigenständige Objekte (Unterobjekte) konstruiert werden und explizit referenzierbar sein.

2.5 Datenhaltung und Zugriff

Eine dreidimensionale Szene besitzt häufig eine hierarchische Struktur. Sie entsteht durch den Konstruktionsprozess der dreidimensionalen Modelle und der Tatsache, dass

es wenige monolithische Objekte gibt. Basis der Konstruktion sind Grundkörper, die sogenannten *Primitiven*. Sie können durch boolesche Operationen verknüpft werden und bilden dadurch komplexere 3D-Modelle. Die endgültige Hierarchie muss innerhalb des virtuellen Labors gespeichert werden. Nach [FD94] kann sie als *gerichteten azyklischen Graphen* repräsentiert werden. Dabei enthalten die Objekte nur Verweise auf Unterobjekte und nicht auf ihr Oberobjekt. In der Datenstruktur zur Speicherung dieser Objekthierarchie muss berücksichtigt werden, dass nicht nur geometrische, sondern auch die zugehörigen applikationsspezifischen Daten wie Farbe und Material gespeichert werden müssen. [FD94] gibt folgende Vorteile zur Speicherung in einer Hierarchie an:

- ☞ Komplexe Objekte können modular konstruiert werden. Dabei werden Bausteine mit unterschiedlichen geometrischen und Darstellungsattributen typischerweise mehrfach eingebettet.
- ☞ Speicherplatz kann gespart werden. Statt der vollständigen Definition des Objektes genügt es, die Verweise auf mehrfach benutzte Objekte zu speichern.
- ☞ Das Modell ist leichter aktualisierbar. Eine Änderung in der Definition eines Bausteins wird automatisch an alle Objekte auf höheren Stufen weitergegeben, die dieses Objekt benutzen.

2.6 3D-Grafikformate

Die 3D-Modelle werden üblicherweise mit professioneller Grafiksoftware erstellt. Die Integration der Modelle in das virtuelle Labor erfordert eine Zwischenspeicherung der Modelle in einem von der Laborumgebung und 3D-Grafiksoftware interpretierbaren Format. Bevor das Modell im virtuellen Labor genutzt werden kann, muss es zwei Transformationsschritte durchlaufen:

① **Export des Modells:** Unterscheidet sich das gemeinsame Format vom nativen Format des 3D-Grafikprogramms, muss das Modell konvertiert und exportiert werden. Dabei kann es zu Verlusten kommen, weil erstens das Grafikprogramm nicht in der Lage ist, alle Informationen des eigenen Formats in das zu exportierende Format zu übersetzen, oder, weil in dem zu exportierenden Format keine zum nativen Format äquivalenten Beschreibungsformen zur Verfügung stehen.

② **Import des Modells:** Das 3D-Modell wird in die Entwicklungsumgebung eingelesen. Hierbei kann es ebenfalls zu Verlusten kommen, wenn die Entwicklungsumgebung nicht alle Attribute (Materialien, Hierarchien) des Formats interpretieren kann.

Ein Ziel der 3D-Visualisierung ist die Verbesserung des Workflows zur Erstellung interaktiver Geräte. Die kann nur gewährleistet werden, wenn der Informationsverlust

beim Export und Import möglichst gering ausfällt, weil ansonsten die verlorengegangenen Information den Modellen in der Laborumgebung erneut hinzugefügt werden müssen. Die von der 3D-Software in das Labor zu übertragenden Daten lassen sich in folgende Kategorien einteilen:

- Objektgeometrien,
- Materialien (Darstellungsattribute),
- Lage und Ausrichtung im Raum,
- Objekthierarchie und
- Verknüpfungen der Objektgeometrien mit den Darstellungsattributen.

Das Grafikformat muss für diese Informationen Beschreibungsformen zur Verfügung stellen. Sie müssen für beide Seiten interpretierbar sein.

2.7 Zusammenfassung

Virtueller Labore lassen sich in die Teile Rahmenhandlung und Experimentierumgebung gliedern. Beide Komponenten stellen teilweise unterschiedliche Anforderungen an die Entwicklung und an die Fähigkeiten der am Entwicklungprozess beteiligten Personen. Die Integration dreidimensionaler Modelle in das virtuelle Labor stellt Anforderungen an die

- Berechnung und Darstellung der Modelle, die durch die Faktoren Qualität und Geschwindigkeit beeinflusst werden,
- Interaktion mit den 3D-Modellen (Manipulation, Kollision, Navigation, Benutzer-System-Schnittstelle)
- Erstellung der 3D-Modelle (geringe Polygonzahl, einfache Modellierungstechniken),
- Verwaltung und Zugriffsmöglichkeiten auf die 3D-Szene (Hierarchie) und
- 3D-Grafikformate zur Übertragung der Modelle aus einer professionellen 3D-Grafiksoftware.

3 Analyse der Entwicklungswerkzeuge

In diesem Kapitel wird systematisch nach geeigneten Werkzeugen gesucht, die die Entwicklung dreidimensionaler Labore unterstützen. Dazu werden vorerst die Begriffe *Entwicklungsumgebungen für Programmiersprachen (IDE)* und *Autorensysteme* definiert sowie deren Nutzen dargestellt. Im darauf folgenden Abschnitt wird ein Kriterienkatalog erstellt, anhand dessen Entwicklungswerkzeuge für die Erstellung dreidimensionaler virtueller Labore unter den für die Arbeit geltenden Rahmenbedingungen analysiert werden sollen. Im Anschluss findet eine konkrete Untersuchung relevanter Werkzeuge statt. Die endgültige Wahl des oder der in dieser Arbeit verwendeten Werkzeuges / Werkzeuge wird in 3.3.4 getroffen.

3.1 Definitionen und Begriffserläuterungen

Zum Erstellen von Softwareprojekten bedarf es heute oft unterstützende Werkzeuge, die den Entwicklungsprozess beschleunigen und dazu beitragen sollen, qualitativ hochwertige Produkte zu erstellen. Die traditionelle Softwareerstellung besteht aus der Programmierung, der Fehlersuche (Debuggen) und dem Übersetzen von Programmen in Binärcode. Diese Aufgaben werden von ausgebildeten Programmieren erfüllt. Als Hilfsmittel benutzen sie meistens eine *IDE*. Eine IDE kann folgendermaßen definiert werden (aus [com], vom Autor übersetzt).

DEFINITION 1 (INTERACTIVE DEVELOPMENT ENVIRONMENT)
(IDE) Ein System, das den Prozess des Schreibens von Software unterstützt. Solch ein System kann einen syntax-gesteuerten Editor, grafische Tools zur Programmeingabe, integrierte Unterstützung für das Übersetzen und Starten des Programms sowie für das Zurückverfolgen von Übersetzungsfehlern zum Quellcode beinhalten.

Äquivalente Begriffe zur IDE sind *Programmierumgebung* und *Umgebung für Programmiersprachen*. Im Zusammenhang mit Programmiersprachen fallen oft die Begriffe *Bibliotheken* und *Frameworks*. Sie unterstützen den Programmierer bei der Lösung anwendungsspezifischer oder allgemeiner Probleme auf rein programmiertechnischer Basis. *Bibliotheken* stellen Funktionalität in Form von schon übersetzten oder

noch als Quellcode vorliegenden Subroutinen dar. Sie können zum eigentlichen Programm hinzugebunden werden. Der Begriff *Framework* wird zunehmend in objektorientierten Systemen verwendet und entspricht einer Menge abstrakter Klassen, die Unterstützung bei der Lösung mehrerer verwandter Probleme bezüglich eines Anwendungsgebietes bieten. Der Zugriff auf den „externen Code“ erfolgt meistens über ein *Application Program Interface (API)*. Es kapselt die eigentlichen Funktionsaufrufe ab und sichert so die Portabilität der Bibliothek.

Im Zuge der Popularität multimedialer Produkte ist eine neue Vorgehensweise zur Erstellung von Software entstanden. Dabei rückt die Programmierung, d. h. das Eingeben und Verifizieren von Programmcode in den Hintergrund. Multimediale Software besteht größtenteils aus einem Netzwerk von Informationsobjekten aus unterschiedlichen Medientypen. Visuelle Techniken erleichtern die Verknüpfung der vielen Informationsobjekte. Reine Präsentationsanwendungen können ganz ohne Programmierung erstellt werden. Andere Anwendungen benötigen weiterhin einen gewissen Teil an Programmierung, um eine komplizierte Ablauflogik zu erstellen oder flexibel auf Interaktionen mit dem Nutzer reagieren zu können. Aufgrund des Wandels von der reinen Programmierung zum Gemisch aus Programmierung und visuellen Techniken, verändert sich ebenfalls die Zusammensetzung der an dem Softwareentwicklungsprozess beteiligten Personen. Das Entwicklungsteam besteht aus Personen mit Programmiererfahrung, meistens Informatiker, und aus Multimedia-Designern, die für die visuellen Erfordernisse einer Multimedia-Produktion verantwortlich sind. In spezielleren Anwendungen, beispielsweise fachspezifischer Lernsoftware, beteiligen sich weitere Personen wie Fachexperten und Didaktiker an dem Entwicklungsprozess.

Diese Veränderungen sorgten dafür, dass eine IDE für Programmiersprachen wie oben beschrieben nicht mehr alleine den Anforderungen eines multimedialen Softwareentwicklungsprozesses gerecht werden konnte. Deshalb wurden in den letzten Jahren Werkzeuge geschaffen, die diesen Prozess besser unterstützen können. Sie werden *Autorensysteme* genannt und von [Bol98] folgendermaßen definiert:

DEFINITION 2 (AUTORENSYSTEME)
Autorensysteme oder -werkzeuge sind graphisch-interaktive Tools, die den technischen Entwicklungsprozess multimedialer Anwendungen mit Hilfe von Konzepten der visuellen Programmierung unterstützen. Sie ermöglichen dabei insbesondere eine graphisch-interaktive Spezifikation der Beziehungen zwischen Medienobjekten.

Diese Definition grenzt den Begriff *Autorensysteme* im klassischen Sinn ab. Neuere Entwicklungen von Autorensystemen legen neben den graphisch-interaktiven Tools ebenfalls Wert auf die Programmierung. Die Abbildung 3.1 veranschaulicht die Aufgaben bei der Erstellung multimedialer Anwendungen und den Einsatz eines Autorensystems.

Idealerweise bieten Autorensysteme Unterstützung in der Programmierung durch beispielsweise Syntax-Hervorhebung, fortgeschrittenen Programmierparadigmen und wiederverwendbaren Klassen, sowie Programmierung für Multimedia-Designer durch einfaches Scripting und parametrisierbaren Verhaltensskripten. Bei der Medienerstellung

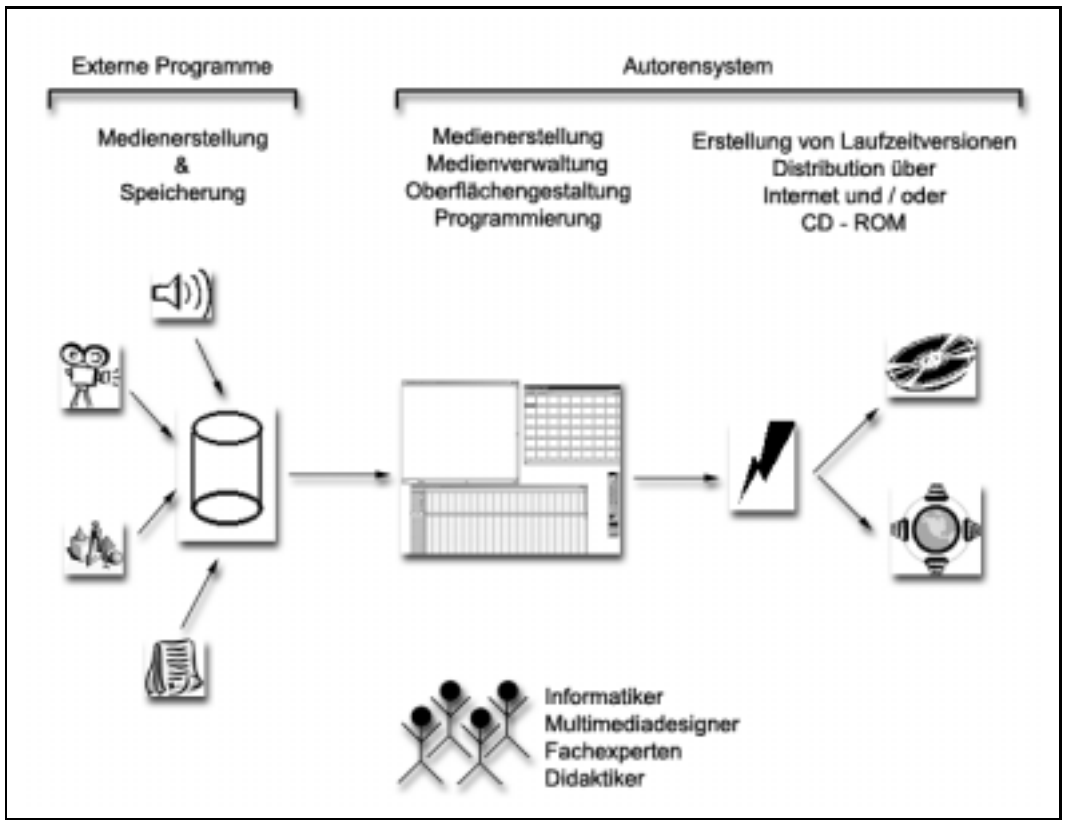

Abbildung 3.1: Aufgaben bei der Erstellung viruteller Labore

und dem -import sollte das Werkzeug den Entwicklern durch einfache grafische Interaktionsmöglichkeiten helfen. Weiterhin vereinfacht ein gutes Entwicklungswerkzeug die Internetanbindung sowie die Erstellung hybrider CD-Rom/Internet Anwendungen. Um auch Nicht-Programmierern einen schnellen Einstieg in die Konzeption von Multimedia-Anwendungen zu ermöglichen, organisieren Autorensysteme den Entwicklungsprozess mit Hilfe von Metaphern. In dem Artikel [BS98] werden u. a. folgende Metaphern genannt: die Stapel-Karten-Metapher, Buchseiten-Metapher, Zeitachsen-Metapher, Flussdiagramm-Metapher, Objekt-Metapher und Skript-Metapher.

In den letzten Jahren hat sich die Kluft zwischen Autorensystemen und Programmierumgebungen verkleinert. Zum einen sind einige Programmierumgebungen benutzerfreundlicher geworden und bieten einige Unterstützung im Bereich Multimedia-Entwicklung. Auf der anderen Seite wurden einige Autorensysteme bezüglich der Programmierfähigkeit verbessert. Ihr Anwendungsspektrum beschränkt sich nicht mehr nur auf die Entwicklung einfacher Präsentationsprodukte, sondern auch zur Entwicklung interaktiver mit Ablauflogik versehener Software. Bei der nachfolgenden Analyse werden daher Werkzeuge aus beiden Gebieten betrachtet.

Ebenfalls betrachtet werden *Beschreibungssprachen*. Beschreibungssprachen bieten Mechanismen zur Zerlegung von Informationen in logische Komponenten, so dass sie von automatischen Systemen entsprechend interpretiert und weiterverarbeitet werden können. Für diese Arbeit sind Beschreibungssprachen für dreidimensionale Wel-

ten interessant. Sie beschreiben eine 3D-Szene über ihren hierarchischen Aufbau und speichern zusätzlich Informationen zu Geometrien, Materialien und Texturen. Alleine wären die Beschreibungssprachen nur zum reinen Austausch von Informationen zwischen 3D-Grafiksoftware und Entwicklungsumgebung interessant. Für einige Beschreibungssprachen existieren jedoch Viewer, welche die Information grafisch darstellen können, also die 3D-Szene rendern. Diese Viewer besitzen manchmal Programmierschnittstellen, damit entsprechend auf Nutzerinteraktion reagiert oder von einem Programm aus die 3D-Szene manipuliert werden kann. Beschreibungssprachen und Viewer stellen keine Alternative zu Autorensystem oder Programmierumgebung dar, sondern können eher als Erweiterung angesehen werden.

3.2 Kriterien zur Wahl der Entwicklungswerkzeuge

Wie die Anforderungen an virtuelle Labore aus Abschnitt 2.1 zeigen, erfordert die Entwicklung virtueller Labore sowohl eine Menge Programmierarbeit speziell für die Experimentierumgebung, als auch einen großen Anteil visueller Techniken zur Verknüpfung von Informationsobjekten in der Rahmenhandlung. Diese müssen durch ein oder mehrere Entwicklungswerkzeuge unterstützt werden. Die Integration dreidimensionaler Modelle erfordert zusätzlich einen hohen Anteil an 3D-Fähigkeiten. Die folgende Auflistung zeigt detailliert die Kriterien auf, welche die Entwicklungswerkzeuge idealerweise unterstützen sollen. Sie ergeben sich hauptsächlich aus den in Kapitel 2 beschriebenen Anforderungen an ein dreidimensionales Labor. Dieser Kriterienkatalog wird verwendet, um die Eignung einer oder mehrer Tools in Kombination zu überprüfen. Einige grundsätzliche Nebenbedingungen sind in der Auflistung nicht explizit enthalten. Beispielsweise können ein 3D-Tool für alle 3d-relevanten Operationen, eine Programmiersprache für die Experimentierumgebung sowie ein Autorensystem für die Erstellung der Rahmenhandlung die Liste der Kriterien optimal abdecken. Wird jedoch berücksichtigt, dass das 3D-Modell sowohl von der Rahmenhandlung als auch von der Experimentierumgebung aus genutzt und dargestellt werden soll, muss das 3D-Tool sowohl von der Programmiersprache als auch von dem Autorensystem aus angebunden werden können, weshalb einige Kombinationen wieder ausgeschlossen werden müssen.

3D-Fähigkeit

- ☞ Import von 3D-Modellen (Vollständigkeit, Komplexität, Unterstützung von Quasi-Standards etc.)
- ☞ Möglichkeit zur Erstellung von einfachen 3D-Modellen (Primitiven)
- ☞ Anzahl der verwendbaren Kamera und deren veränderbare Parameter
- ☞ Anzahl der verwendbaren Lichtquellen, Anzahl der und Beleuchtungsarten und die Möglichkeit die Parameter der Lichtquellen zu verändern
- ☞ Schattierungsalgorithmen (möglichst Gouraud- oder

Flatshading, zusätzlich Drahtgitter, dynamischer Wechsel für ausgewählte Objekte)

- ☞ Hardware-Unterstützung
- ☞ Projektionsarten für Texturen / Materialien
- ☞ Verwendung von Umgebungsmaps
- ☞ Geschwindigkeit der Darstellung
- ☞ einfache Interaktionsmechanismen
  - ➢ Rotation
  - ➢ Translation
  - ➢ Skalierung
- ☞ Kollisionsentdeckung
  - ➢ Bounding Boxes
  - ➢ Polygonstruktur
- ☞ Hilfestellungen bei der Interaktion im dreidimensionalen Raum

Programmierbarkeit

- ☞ Paradigma
  - ➢ objektorientiert
  - ➢ imperativ
  - ➢ funktional
- ☞ Möglichkeit zur Anbindung und Erweiterung des GENLAB-Frameworks
- ☞ Angebot vorhandener Klassen/Skripte/Prozeduren/Funktionen
- ☞ Erweiterbarkeit
- ☞ Einbindung von Plug-Ins
- ☞ Kontrollierbarkeit (Möglichkeit zur dynamischen Manipulation von Objekten, Eventhandling etc.)

Codeerzeugung

- ☞ Plattformunabhängigkeit (Betriebssysteme, Webbrowser)
- ☞ Systemvoraussetzungen (Speicher, Prozessor)
- ☞ Geschwindigkeit der Runtime-Version
- ☞ Standalone-Version

Medienverwaltung und -erstellung

- ☞ Anzahl der unterstützten Medienformate
- ☞ Art der Verwaltung (interne Verwaltung, Datenbank Filesystem)
- ☞ Möglichkeiten der Medienerstellung und -bearbeitung
  - ➢ Text und Hypertext
  - ➢ Grafiken
  - ➢ Audio
  - ➢ Video
  - ➢ Animationen
- ☞ Zugriffskontrolle und Versionskontrolle

Bedienbarkeit

- ☞ einfache interaktive grafische Bedienung für Nicht-Informatiker insbesondere in der Medienerstellung und -verwaltung
- ☞ einfache Programmierung von „kleinen“ Skripten (Behaviours)
- ☞ Tools für die komplexe Programmierung
  - ➢ Debugger
  - ➢ Editor mit Syntaxhighlighting
  - ➢ automatischer Projektverwaltung
  - ➢ Hierarchiebrowser

Sonstiges

- ☞ Anschaffungskosten
- ☞ Kosten zur Distribution
- ☞ Bedingungen zur Distribution

3.3 Analyse

In der Analyse wurden Programmiersprachen und deren Entwicklungsumgebungen, Bibliotheken, Szenenbeschreibungssprachen und Autorensysteme mit möglichen Erweiterungen wie PlugIns oder Xtras berücksichtigt. In die Auswahl der näheren Analyse wurden nur Produkte aufgenommen, die mehrere Kriterien aus Abschnitt 3.2 erfüllen konnten. Andere ebenfalls betrachteten Werkzeuge werden nicht aufgelistet. Tabelle 3.1 zeigt die genauer untersuchten Entwicklungswerkzeuge im Überblick.

In den nachfolgenden Abschnitten werden die charakteristischen Merkmale der einzelnen Produkte sowie deren Vor- und Nachteile in Bezug auf die Erstellung dreidimensionaler Labore erörtert. Der Abschnitt 3.3.4 vergleicht die in Frage kommenden Alternativen miteinander und stellt das geeigneteste bzw. die geeignetesten Entwicklungswerkzeuge für die Konzeption dreidimensionaler virtueller Labore heraus.

3.3.1 Die Autorensysteme im Detail

Macromedia Director, Version 8.0

Macromedia DIRECTOR ist das populärste unter den Multimedia-Autorensystemen. DIRECTOR ist nicht auf die Entwicklung bestimmter Multimedia-Produkte wie CBT-Systeme, Datenbankanwendungen oder Präsentationsprodukte spezialisiert, sondern eher Generalist. Der Autor ist in der Lage per Drehbuch seine Medien auf der Bühne zu platzieren. Es eignet sich hervorragend für die Erstellung von Animationen. Dazu können sowohl Bitmaps wie auch das Vektorgrafikformat FLASH verwendet werden. Alle gängigen Formate für 2D-Grafik, Sound und Video werden unterstützt. Eine externe Schnittstelle erlaubt das Einbinden sogenannter meist in der Programmiersprache C geschriebenen Xtras, die DIRECTOR in seiner Funktionalität erweitern. Wegen der Beliebtheit dieses Autorensystems sind viele durch Drittanbieter entwickelte Xtras sowohl frei als auch kommerziell verfügbar. Seit der Version 6.0 wird das QUICKDRAW

Produktname	Firma, Organisation	Website
Autorensysteme		
Director	Macromedia	http://www.macromedia.com/software/director/
iShell	Tribeworks	http://www.tribeworks.com/home/
Pulse3D	Pulse Entertainment	http://www.pulsenetwork.com/
Toolbook	click2learn.com	http://www.click2learn.com
Programmiersprachen, Bibliotheken		
Delphi	Borland / Inprise	http://www.borland.com/delphi/
Java / Java3D	Sun	http://www.sun.com
Mathengine	Mathengine	http://www.mathengine.com
Quicktime	Apple Computer	http://www.apple.com/quicktime/
Visual Studio 6.0	Microsoft	http://www.microsoft.com/germany/msdn/vstudio/
Beschreibungssprachen, Sonstige Tools		
Metastream / VET	Metacreations / Viewpoint Corporation	http://www.viewpoint.com/
Strata 3D Cult3D	Strata Software Cycore	http://www.strata3d.com/ http://www.cult3d.com/
VRML / X3D	Web3D Consortium	http://www.vrml.org/

Tabelle 3.1: Die getesteten Entwicklungswerkzeuge im Überblick

3D-Xtra mit Director ausgeliefert. Auf diesem Wege unterstützt Macromedia selbst den Import und die Darstellung dreidimensionaler Modelle (s. hierzu weiter unten). Weiterhin ist die Einbindung von ActiveX-Komponenten möglich.

Ein großer Vorteil gegenüber vielen anderen Autorensystemen ist die in den letzten Jahren stark verbesserte objektorientierte Skriptsprache LINGO. Mit ihr können ausgebildete Programmierer komplexe Abläufe implementieren und wiederverwendbaren Code erzeugen. Der Multimedia-Designer kann dank der einfach gehaltenen Skriptsprache kleine Behaviours erstellen oder sich der vielen parametrisierbaren Skripte aus den mitgelieferten Bibliotheken bedienen. DIRECTOR bietet Möglichkeiten zur Anbindung der Applikation an das Internet an. Die Distribution wird sowohl über CD-Rom als auch über das Internet durch die weitverbreitete SHOCKWAVE-Technologie ermöglicht und ist genau wie die Autorenumgebung auf Macintosh und Windows Betriebssystemen lauffähig. Dank der Offenlegung des Shockwave-Flash Formats entwickeln einige freie Entwickler an einer Abspielmöglichkeit auf Unix-Systemen.

Director Erweiterungen

☞ **3D-Dreams von Shells Interactive, Version 2.5 RC 2**
3D-DREAMS ist eine Erweiterung für das Autorensystem DIRECTOR. Das Produkt befindet sich in einer frühen Entwicklungsphase. Das Xtra bindet seine

Produktname	Firma, Organisation	Website
Director Erweiterungen		
3D-Dreams	Shells Interactive	http://www.doitin3d.com/3ddreams/
3D-Groove	The Groove Alliance	http://www.3dgroove.com
Dave's 3D-Engine	Dave Cole	http://www.dubbus.com/devnull/
Focus3D	Focus3	http://www.umminger.com/focus3/
Openspace For Director	Virtus	http://www.virtus.com/products/ software_specsapps_data.html
Quickdraw 3D	Macromedia	http://www.macromedia.com/ software/director/
T3D-Engine	T3D@THEBURROW	http://www.theburrow.co.uk/ t3dtesters/
Virtools NeMo	Virtools SA	http://www.next-url.com/

Tabelle 3.2: Macromedia Director Erweiterungen im Überblick

eigene Oberfläche zum Import und zur weiteren Manipulation in DIRECTOR ein (siehe Abbildung 3.2). So können beispielsweise die Koordinaten, Winkel und Größe der 3D-Objekte innerhalb der Umgebung bestimmt werden. Bei Veränderung der Modelle durch eine andere Software findet eine automatische Synchronisation statt. Die Objekte können hierarchisch in eine Szene eingegliedert werden. Die hierarchische Anordnung kann während der Laufzeit erfolgen. Die Wurzel der Szene stellt das *Universum* dar. Davon abgleitet lassen sich mehrere Welten ableiten. Jede Welt enthält mindestens eine Kamera und eine Lichtquelle. Diese können aber auch anderen Objekten untergeordnet sein. Eine dynamische Änderung der Kamera- und Beleuchtungsparameter ist möglich. 3D-DREAMS unterstützt mehrere Kameras und verschiedene Beleuchtungsarten. Alle Objekte lassen sich via LINGO einfach manipulieren. Als Importformate geben die Entwickler VRML, METASTREAM, X-OBJECTS und das 3D STUDIO-Format an. Der Import von 3ds-Dateien schlägt in der neuesten Version jedoch fehl. Einige Objekte werden auf den Ursprung des Weltobjektes zurückgesetzt und ihre Winkel neu ausgerichtet. Nach Angaben der Entwickler soll das Problem in der nächsten Version behoben werden. Bis dahin solle man sich mit den anderen Formaten begnügen. Das einzige z. Z. taugliche Format ist das X-OBJECTS. Die anderen Formate kommen mit komplexeren Objekten nicht zurecht. Leider bietet X-OBJECTS nicht die Möglichkeit, Hierarchien abzuspeichern. Soll trotzdem 3D-DREAMS verwendet werden, muss die Objekthierarchie der einzelnen Objekte innerhalb der 3D-DREAMS-Umgebung wieder hergestellt werden. Das Produkt bietet neben der reinen Software-Darstellung ebenfalls eine Hardware-Unterstützung an. Das Xtra verwendet zur Kollisionsentdeckung Bounding-Boxes. Schattenwurf ist nicht möglich. Als Schattierungsalgorithmen werden Flatshading und Drahtgitterdarstellung angeboten.

☞ **3D-Groove**
Das DIRECTOR Xtra 3D-GROOVE konnte leider nur an Demos „getestet" werden. Es wird z. Z. nicht veröffentlicht, sondern in der Herstellerfirma THE

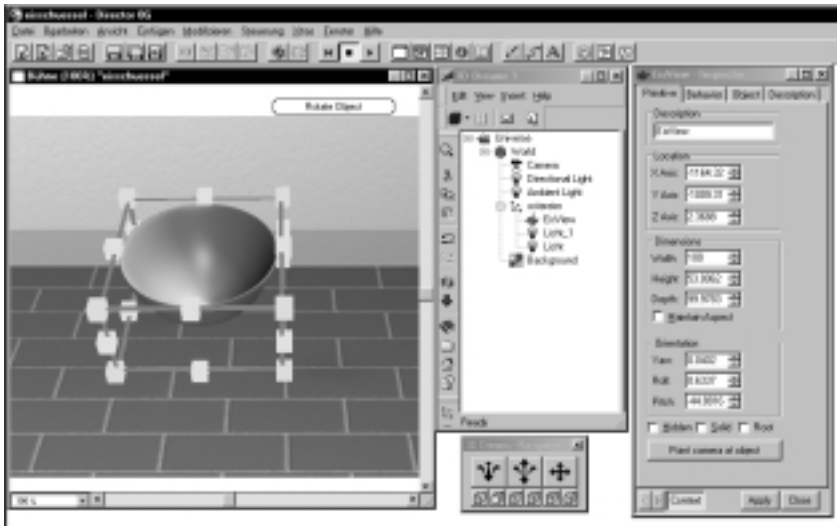

Abbildung 3.2: 3D DREAMS in Director 8 integriert

GROOVE ALLIANCE verwendet. Ein Development Kit soll erst in einigen Monaten herausgegeben werden. Preise wurden von den Entwicklern nicht genannt. Die Beispielfilme lassen Schattenwurf und genaue Kollisionserkennungsmechanismen erkennen. Die Objekte selbst sind nur äußerst einfach dargestellt, die Interaktionsmöglichkeiten in den Demos sind gering. Es können mehrere Kameraeinstellungen und Lichtquellen verwendet werden. Die Darstellungsqualität deutet auf Flatshading.

☞ **Dave's 3D-Engine Verion 7.6**
Die Shareware DAVE'S 3D ENGINE ist komplett in DIRECTORS Skriptsprache LINGO geschrieben. Der Code ist komplett verfügbar und nach eigenen Wünschen änderbar. Eine Einbindung über die Xtra-Schnittstelle ist nicht notwendig. Es stehen Operationen zur Rotation, Translation und Skalierung zur Verfügung. Zusätzlich zu der Darstellung der Objekte mittels Sprites, also Bitmaps, gibt es die Möglichkeit die Objekte mittels DIRECTORs Quad-Darstellung. Hier werden die Objekte durch Polygone über Punktmengen definiert. Die Polygone können mit einer Textur belegt werden, jedoch wird diese nicht perspektivisch auf das Objekt gemapped. Beleuchtungsmodelle wurden noch nicht integriert. Es gibt z. Z. nur eine Kamera. Die Funktionsaufrufe zur Darstellung und Manipulation der 3D-Objekte sind eng an die der 3D API OPENGL angelehnt. Als Import-Formate werden die eigenen D3D- und P3D-Formate sowie das WAVEFRONT-Format OBJ unterstützt. Insgesamt ist DAVE'S 3D-ENGINE ein interessanter Ansatz zur 3D-Darstellung in DIRECTOR. Zur Visualisierung dreidimensionaler Umgebungen ist das Produkt noch lange nicht geeignet. Der Import von Wavefront-Objekten weist oft Fehler auf und funktioniert für einige Objekte gar nicht. Die Erstellung der Objekte durch manuelle Eingabe der Punktmengen ist viel zu mühsam und zu zeitaufwendig. Eine Kollisionserkennung ist nicht integriert.

☞ **Focus 3D**
FOCUS 3D ist ein DIRECTOR-Xtra zur Einbindung von QuickDraw3DModellen. Mehrere Modelle können zu einem Composite-View kombiniert werden. Sie können neu koloriert und texturiert werden. Die Szene ist mit einer beweglichen Kamera ausgestattet, mehrere Lichtquellen lassen sich hinzufügen. Das Xtra ist nur auf Macintosh-Rechnern ausführbar und seine Entwicklung wurde bereits 1997 eingestellt. Weitere Untersuchungen wurden deswegen nicht vorgenommen.

☞ **Openspace 3D for Director**
Ein Test von OPENSPACE 3D war leider nicht möglich, da weder eine Demo-Version zur Verfügung stand noch der Support auf Anfragen zum Produkt antwortete. Die folgenden Informationen stammen von der Website des Herstellers. In erster Linie ist OPENSPACE 3D eine Standalone-Umgebung zur Erstellung komplexer virtueller Szenarien. Sie wurde speziell zur Integration in Macromedia DIRECTOR entworfen. In Zukunft sollen auch andere Produkte wie TOOLBOOK und AUTHORWARE und Programmiersprachen wie C++ und Visual Basic unterstützt werden. Die erstellten Szenarien lassen sich per ActiveX-Control in DIRECTOR einbinden und vollständig durch LINGO manipulieren (laut LINGO-Referenz von OPENSPACE). Mehrere Kameras, deren Projektionsarten auswählbar sind, können in die Szene integriert werden. Als 3D-Importformat werden 3D-STUDIO- und DXF-Dateien unterstützt. Sowohl Software- als auch Hardware-Rendering ist möglich. Zur Schattierung dienen Flat- und Gouraud-Shading. Mehrere in Farbe und Art parametrisierbare Lichtquellen können die Szene erhellen. Bisher wird nur die Windows-Plattform unterstützt. Eine Macintosh-Version ist angeblich in Planung.

Die Produktbeschreibung klingt nach einer vielversprechenden Lösung zur Integration von 3D in DIRECTOR und sollte falls es denn verfügbar wird berücksichtigt werden.

☞ **Quickdraw 3D Xtra von Macromedia**
Das seit der DIRECTOR Version 6.0 mitgelieferte Xtra bietet nur grundlegende 3D-Funktionalität auf der Basis von Apples QUICKDRAW 3D-Format. Seitdem ist das Xtra nicht mehr verbessert worden. Texturen werden beim Import nicht berücksichtigt. Die Nachtexturierung über Castmember hat beim Testen fehlgeschlagen. Die Darstellungsqualität und -geschwindigkeit ist nicht ausreichend. Beleuchtungsmodelle fehlen.

☞ **T3D Engine Version 0.9**
T3D ist eine ähnlich wie DAVE'S 3D-ENGINE komplett in LINGO geschriebene Engine zur Darstellung dreidimensionaler Modelle in DIRECTOR. Der Quellcode ist frei verfügbar. Es lassen sich sehr einfache 3D-STUDIO-Dateien ohne Berücksichtigung der Objekthierarchien einlesen. Bis zu acht Lichtquellen, parametrisierbar über Farbe, Stärke, Position und Richtung, können in die Szene integriert werden. Eine einfache Kollisionsentdeckung auf der Basis von Bounding-Boxes ist vorhanden. Bei Tests kam es mehrfach zu Fehlern bei der

Kamerapositionierung. Die Spriteverwaltung muss manuell durchgeführt werden, d. h. es müssen in DIRECTOR die für ein Objekt benötigten Spritekanäle reserviert werden.

☞ **Virtools NeMo**
Dieses Werkzeug kann eher als Entwicklungsumgebung für 3D-Spiele denn als Autorensystem für multimediale Anwendungen charakterisiert werden. Es dient der Erstellung dynamischer und interaktiver dreidimensionaler Welten. Es können 3D-STUDIO-Szenen und -Animationen, LIGHTWAVE-Modelle und -Animationen über das DIRECTX-Format und WAVEFRONT / MAYA-Modelle importiert werden. Neben der alleinigen Softwareberechnung unterstützt das Produkt die Hardwareberechnung von DIRECTX-, OPENGL- und GLIDE-Anweisungen. Behaviours aus den mitgelieferten Bibliotheken können grafisch miteinander kombiniert werden. Komplexe Routinen können in C++ programmiert und eingebunden werden. Dazu wird allerdings das teure NEMO-Development Kit benötigt. Insgesamt richtet sich das Tool an versierte Spiele-Programmierer. Es besteht die Möglichkeit, VIRTOOLS in DIRECTOR als ActiveX-Komponente einzubinden und per LINGO-Befehle zu kontrollieren. Der Kontroll- und Datenfluss zwischen DIRECTOR und der ActiveX-Komponente ist äußerst unbefriedigend.

iShell von Tribeworks, Version 2

ISHELL ist ein auf die Erstellung von Internetapplikationen und Kioske ausgerichtetes Autorensystem. Die Software überzeugt durch ein große Menge von Importformaten und eine sehr gute QUICKTIME-Integration. Präsentationen lassen sich schnell durch grafisch-interaktive Werkzeuge erstellen. Eine objektorientierte Programmierumgebung und ein Debugger unterstützen den Programmierer in der Implementation wiederverwendbarer einfacher Skripte größtenteils durch visuelle Programmierung. Komplexe Abläufe sind wegen des geringen Befehlsvorrates und fehlender Kontrollstrukturen nur schwer programmierbar. Zur Erweiterung können Drittanbieter Module erstellen, die über eine einheitliche Schnittstelle in ISHELL integriert werden. Die Autorenumgebung und das Laufzeitsystem können auf Macintosh- und Windows-Rechnern installiert werden. 3D-Fähigkeiten gibt es weder durch das ISHELL selbst, noch sind Module von Drittanbietern bekannt.

Pulse3D von Pulse Entertainment

PULSE3D ist ein eigenständiges auf Web-Distributionen spezialisiertes Autorensystem. Ungewöhnlich ist seine Lizenzierung. Es fallen erst Kosten an, sobald eine mit „Pulse-Inhalt“ versehene Webseite in das Internet gestellt wird. Die Kosten betragen zwischen $2500,- und $10000,- jährlich. Es besitzt eine einfache Skriptsprache namens JOESCRIPT zur Kontrolle aller einfachen Interaktionen, jedoch nicht zur Implementierung komplexer Abläufe. Eine Integrierung in das GENLAB-Framework ist damit ausgeschlossen. An 3D-Formaten werden VRML, DXF und 3D-STUDIO unterstützt. Es gibt dynamische Beleuchtungsmodelle und Texturierung. Das Produkt

kann Bump- und Reflektions-Maps verarbeiten. Auf Wunsch kann eine Kantenglättung aktiviert werden. Eine Hardwareunterstützung und eine dynamische Polygonreduktion wird zur Geschwindigkeitssteigerung angeboten. Es lassen sich dreidimensionale Animationen und durch Inverse Kinematik modellierte Charaktere erstellen. Die Autorenumgebung kann nur auf Windows-Rechnern installiert werden. Mit PULSE3D entwickelte Produkte sind sowohl unter Windows im Internet Explorer und Netscape sowie auf Macintosh-Rechnern im Netscape abspielbar. Das Autorensystem macht insgesamt einen sehr ausgereiften Eindruck. Durch die Spezialisierung auf 3D-Authoring, werden die konventionellen Funktionen wie 2D-Animation und der Import von Videos vernachlässigt.

Toolbook Instructor 7.2

TOOLBOOK ist nach DIRECTOR das meist verbreiteste Autorensystem. Mittlerweile hat es sich auf die Erstellung von Online-Kursen spezialisiert und bietet dort gegenüber seinen Konkurrenten einige Vorteile wie beispielsweise Kursmanagement. Das Spektrum an 2D-Grafik Unterstützung ist immens. So kann TOOLBOOK insgesamt 19 verschiedene Bildformate einlesen. Es besitzt Schnittstellen zu PARADOX und DBASE Datenbanken, sowie direkten Export nach DHTML. Das Autorensystem ist nahezu doppelt so teuer wie DIRECTOR, bietet aber hinsichtlich der Programmierumgebung mit der objektorientierten Skriptsprache OPENSCRIPT gegenüber LINGO keine Vorteile. Die Syntax gleicht traditionellen Skriptsprachen. Sie ist an der natürlichen Sprache angelehnt und eignet sich daher gut für Multimedia-Designer, aber nicht für erfahrene Programmierer. Zur Erweiterung der TOOLBOOK-Funktionalität wird lediglich eine ActiveX-Schnittstelle sowie die Möglichkeit der Einbindung von dynamischen Bilbiotheken (Dlls) angeboten. Das Produkt selbst unterstützt keine 3D-Visualisierung. Erweiterungskomponenten sind nicht bekannt.

3.3.2 Die Programmiersprachen und Bibliotheken im Detail

Delphi 5

Die objektorientierte Programmiersprache DELPHI besitzt durch die zugehörige IDE die Fähigkeit, Oberflächen größtenteils durch visuelle Programmierung, d. h. durch Drag & Drop-Mechanismen zu erstellen. Dazu kann sich der Entwickler der VCL (Visual Component Library) bedienen, die z. Z. 200 verschiedene Komponenten wie HTML und XML Darstellung, Integration von Microsoft Office Applikationen, Einbindung von Graphiken und Datenbankanbindungen umfasst. Die Programmierfähigkeiten sind denen der reinen Autorensystemen überlegen. Es sind vom Editor mit Syntax-Highlighting über Debugger und Hierarchiebrowser alle für Programmierer wichtigen Eigenschaften vorhanden. Es fehlt an Multimedia-Funktionen. Zwar können Grafiken, Sound und Texte in ein Produkt integriert werden, jedoch fehlen für Autoren u. a. Metaphern wie Drehbuch- oder Buchseitendarstellung. Animationen kön-

nen mit Delphi nicht visuell erstellt werden. 3D-Graphik kann nur durch Drittanbieter-Produkte über die ActiveX-Schnittstelle eingebunden werden.

Java / Java3D

Sun stellt mit der JAVA-2-Erweiterung JAVA 3D die Möglichkeit zur Verfügung, plattformunabhängige dynamische dreidimensionale Welten zu erzeugen. JAVA 3D ist eine API zur Implementierung interaktiver 3D-Szenen und Objekte. Durch Programmierung können Universen, Szenen und Objekte, in einer Baumstruktur gegliedert, generiert werden. Die Klasse `Shape3D` kümmert sich um die Erstellung und Verwaltung der dreidimensionalen Objekte. Ihnen können dynamisch Materialien (Farbe, Texturen und Transparenz) zugewiesen werden. Manipulationsfunktionen wie Rotation und Translation stellt die API zur Verfügung. Das Kamera-Konzept wird in JAVA 3D verallgemeinert, sodass die erstellten Welten nicht nur über einen üblichen Monitor sondern auch beispielsweise über Brillen mit zwei LC-Displays und Sensoren für Kopfposition perspektivisch korrekt betrachtet werden können. Zur Beleuchtung der Szene können verschiedene Lichtquellen-Objekte erstellt werden, die von der Blattklasse `Light` bzw. deren Spezialisierungen `DirectionalLight, PointLight, Spotlight` oder `AmbientLight` abgeleitet für entsprechendes realistisches Aussehen sorgen. Ein großer Vorteil in der Verwendung von JAVA 3D liegt in der Verbindung mit den restlichen Java-2-Fähigkeiten. So ist es möglich komplexe Interaktionen und Abläufe zu programmieren. Netzwerkfunktionalität ist bereits in Java integriert. Wie Sigurd Elfreich in [Elf99] zeigt, kann das GENLAB-Framework nach Java übertragen werden. Damit würde eine 3D-Erweiterung des Frameworks machbar. Leider fehlt es an gut konzipierten Entwicklungsumgebungen. Es gibt (noch) keine Werkzeuge zum Import von 3D-Formaten. Alle 3D-Objekte müssen durch reine Programmierung erzeugt werden. Das bedeutet nicht nur einen erheblichen Zeitaufwand sondern auch qualitativ schlechteres Aussehen. Die Erstellung von Animationen, die Integration und Synchronisation von Medienobjekten lassen sich in Autorensystemen wesentlich einfacher und schneller durchführen.

Mathengine

Die MATHENGINE stellt ähnlich wie JAVA3D eine API dar über die auf mehreren Bibliothen zugegriffen werden kann. Sie können in C++-Programmen eingebunden werden und bieten hervorragende Funktionen für die Implementierung physikalischer Eigenschaften wie Gravitation, Flugbahnberechnung und Kollisionserkennung. Es werden die Plattformen Windows, Linux, SGI Irix und Playstation 2 unterstützt. Der Source-Code wird mitgeliefert. Wie bei allen APIs kann auch die MATHENGINE nur durch Programmierung genutzt werden. Graphische Tools zur Erstellung oder zum Import der 3D-Szene und Objekteigenschaften gibt es nicht. Selbst einfache Szenen werden schnell komplex wie das Listing im Anhang A demonstriert. Damit eignet sich dieses Toolkit eher für die wesentlich aufwendigere 3D-Spiele Entwicklung und nicht zur Erstellung virtueller Labore.

Quicktime 5

QUICKTIME stellt Zugriffsmöglichkeiten auf seine Technologie über eine API her. Anders als in „echten“ 3D-Programmen setzen sich QUICKTIME-Objekte nicht aus Polygonen zusammen, sondern aus bereits berechneten Bitmaps, die hoch komprimiert und zu einem Objekt zusammengefügt werden können. Bei der Wiedergabe bedarf es nur noch der Dekomprimierung der Bitmaps und einfachen perspektivischen Berechnungen. Die QUICKTIME-Technologie ermöglicht eine photorealistische aber unflexible Darstellung. Interaktionsmöglichkeiten sind äußerst gering und hängen von den zur Erstellung des Objektes verwendeten Einzelbildern ab. Üblicherweise lässt sich verschieben, rotieren und auf der Basis von Bitmaps nur eingeschränkt und unbefriedigend skalieren. Durch die Verwendung anderer als die für eine Rotation notwendigen Bitmaps kann die üblicherweise für Rotationen genutzt Mausbewegung zur Darstellung anderer Eigenschaften genutzt werden. So kann beispielsweise durch entsprechender Mausbewegung eine Tür einer Mikrowelle geöffnet oder zeitliche Abläufe visualisiert werden. Mehrere QT-Objekte können in einer virtuellen Szene angeordnet werden. Durch sogenannte Hotspots kann der Anwender zwischen verschiedenen Szenen wechseln. Alleine bietet QUICKTIME nicht annährend genügend Funktionalität für die dreidimensionale Darstellung eines Labors. Viele Autorensysteme unterstützen die Einbindung von QUICKTIME-Objekten. Durch das im Gegensatz zur vektor-basierten Darstellung hohe Datenvolumen der QT-Objekte dauert das Laden einer Szene sehr lange. QUICKTIME-Objekte besitzen keinerlei Tiefeninformation. Kollisionen können nicht erkannt und Interaktionen zwischen Views können auf visueller Ebene nicht realisiert werden.

Visual Studio 6.0

Das von Microsoft entwickelte Toolpaket zur Entwicklung von Software beinhaltet eine für Programmierer sehr gute IDE für die Programmiersprachen Visual C++, Visual J++, Visual Basic und Visual Fox Pro. Sie erlaubt die visuelle Programmierung von einfachen Benutzungsoberflächen. Dem Programmierer werden neben Debugger, Hierarchiebrowser, Tools zum Datenbankzugriff und Code-Completion die Versionsverwaltung Source-Save bereitgestellt. Allerdings fehlen jegliche Werkzeuge zum visuellen Multimedia-Authoring, d. h. zur Erstellung von Animationen, Verknüpfung und Import von Informationsobjekten etc. VISUALE STUDIO 6.0 ist daher nicht für die Produktion multimedialer Software geeignet.

3.3.3 Beschreibungssprachen, sonstige Tools

Metastream / Viewpoint Experience Technology

Die Firma METASTREAM, Tochter von METACREATIONS, wurde im Oktober 2000 zu VIEWPOINT CORPORATION umbenannt. Im November 2000 wurde die Firma

METACREATIONS in VIEWPOINT CORPORATION eingegliedert. Die METASTREAM-Technologie (MTS) wurde in VIEWPOINT EXPERIENCE TECHNOLOGY (VET) umbenannt. Da der bisherige Name z. Z. noch geläufiger ist, wird dieser in der Arbeit weiterverwendet. Ähnlich wie VRML ist METASTREAM eine Sprache zur Beschreibung von dreidimensionalen Objekten und Szenen. Diese können innerhalb eines Webbrowsers abgespielt werden. Innerhalb der Szene können HTML und XML integriert werden. Das Produkt verfügt aber über keine eigene Skriptsprache. Skripte können nur über das Hostsystem, also dem Webbrowser, mit beispielsweise JavaScript erstellt und in die Szene eingebunden werden. Im Gegensatz zu VRML ist METASTREAM nicht auf die Navigation innerhalb virtueller Umgebungen spezialisiert, sondern auf die dreidimensionale Präsentation einzelner Objekte. Da dies weniger rechenintensiv ist, kann die Rechenleistung für eine qualitative hochwertige Darstellung der 3D-Modelle verwendet werden. So werden Schattenwurf, Umgebungsmaps und prozedurale Texturen zur Visualisierung verwendet. Die Interaktionsmechanismen erlauben nur eine einfache Manipulation der Objekte. Neben Mouseevents können anders als bei VRML auch Tastaturevents verwertet werden. Die METASTREAM-Technologie gewinnt seit kurzem an Popularität und würde sich in Verbindung mit anderen Autorensystemen zur Darstellung dreidimensionaler Produkte eignen. Bis jetzt sind allerdings noch keine solchen Systeme bekannt.

Strata 3D und Cult3D

STRATA 3D ist eine einfache freie 3D-Software zur Erstellung dreidimensionaler Modelle. Fortgeschrittene Funktionen wie Inverse Kinematik oder Spezialeffekte können durch Zusatzmodule erworben werden. Neben den Exportformaten DXF und VRML bietet die Software das Format CULT 3D DESIGNER FILE an. Dieses Format ist ähnlich wie VRML eine Beschreibungssprache für dreidimensionale Welten, die in einem Webbrowser über ein PlugIn abgespielt werden können. Die Herstellerfirma von CULT3D stellt zur Erstellung der Szenen grafische Tools (Editoren) zur Verfügung. Mit ihnen lassen sich einfache eventbasierte Interaktionen vollkommen ohne Programmierung realisieren. Kollisionen werden nicht erkannt. Über eine Java-Schnittstelle können Java-Applets angestoßen werden. Umgekehrt können von Java-Applets aus einfache Objektmanipulationen durchgeführt werden. Die Darstellungsqualität ist besser als die der VRML-Renderer, reicht jedoch nicht an die von METASTREAM heran. STRATA 3D ist das einzige bekannte Produkt zur Erstellung von Cult3D-Szenen.

VRML 2 / 97

VRML selbst ist weder eine Programmiersprache noch ein Autorensystem. Es stellt eine Beschreibungssprache zur Darstellung virtueller dreidimensionaler Welten dar. VRML besitzt keine Sprachelemente zur Programmierung von logischen Abläufen. Es bietet nur geringe Interaktionsmöglichkeiten. Zwischen Szenen kann gesprungen oder neue Informationen (z. B. Texturen) von einer festgelegten URL geladen werden. Zur die Interaktion mit Objekten innerhalb einer Szene kann bis zum heutigen Zeitpunkt nur die Maus verwendet werden. Tastatureingaben werden nicht erkannt. Mittels

Script-Knoten können Programme anderer Sprachen angestoßen werden. Da VRML bis zum Zeitpunkt der Aufnahme zur ISO-Standardisierung nur für die Darstellung im WWW, also innerhalb eines Browser, ausgelegt war, können die von Script-Knoten angestoßenen Programme praktisch nur in einer dem Browser interpretierbaren Sprache, beispielsweise JavaScript, VBscript oder Java geschrieben sein. Erst später wurde die *EAI* (External Authoring Interface) entwickelt. Diese Schnittstelle stellt mittels Java-Klassen einen Zugang direkt zum VRML-Szenegraphen außerhalb des WWW-Browsers her. Auf diese Weise können VRML-Szenen in Java-Applets eingebettet werden. Abbildung 3.3 zeigt die Programmierschnittstellen zu VRML. JAVA3D bietet

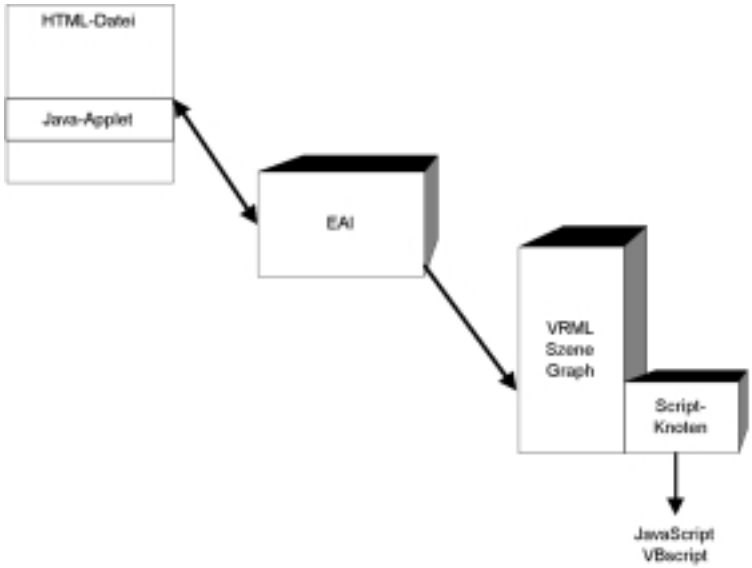

Abbildung 3.3: Programmierschnittstellen zu VRML-Szenen. Aus [KRSD98].

mittlerweile mehr Fähigkeiten zur Darstellung dreidimensionaler Welten und zur Interaktion mit 3D-Objekten. Ein Vorteil der Verwendung von VRML-Szenen innerhalb von Java Applikationen besteht in der Möglichkeit, die Szenen durch grafische Editoren ohne Programmierung erstellen zu können. Die meisten 3D-Grafikprogramme sind in der Lage, nach VRML zu exportieren und können dann in VRML-Editoren weiterverarbeitet werden. Die Beschreibungssprache wurde für die Darstellung im WWW geschaffen. Ein Ziel der VRML-Organisation war es deshalb, die zu übertragenden Dateien möglichst klein zu halten. Dies führte zu einer von vielen Leuten bemängelten qualitativ geringwertigen Visualisierung. Nach einer Umfrage von Cindy Ballreich in [Bal99] waren 90 Prozent mit der Darstellungsqualität von VRML-Szenen unzufrieden. Im Vordergrund von VRML steht nicht die Interaktion mit Objekten, sondern die Navigation in virtuellen Welten. Diese erfordert eine sehr hohe Rechenleistung. Zur performanten Darstellung werden in VRML-Playern nur einfache Schattierungsalgorithmen verwendet.

3.3.4 Ergebnis der Analyse

Insgesamt wurden 20 Werkzeuge, die in der Entwicklung dreidimensionaler Labore hilfreich sein könnten, näher untersucht. Dabei wurde festgestellt, dass nicht ein einzelnes Werkzeug mächtig genug ist, in allen anfallenden Aufgaben Unterstützung zu bieten. Folglich muss eine Kombination mehrerer Entwicklungswerkzeuge eingesetzt werden. Die Vor- und Nachteile der Tools wurden im vorigen Abschnitt beschrieben. Beim Einsatz mehrerer Werkzeuge muss eine nahtlose Produkterstellung gesichert bleiben. Aus der Analyse ergeben sich zwei denkbare Lösungen. Die erstere setzt sich aus den Werkzeugen Macromedia DIRECTOR 8 und 3D-DREAMS zusammen. Diese Kombination bietet eine sehr gute Anbindungs- und Erweiterungsmöglichkeit des GENLAB-Frameworks, sowie die Nutzung aller durch DIRECTOR mitgelieferten Funktionen zum Multimedia-Authoring. Darüber hinaus können die 3D-Modelle durch professionelle Grafiksoftware erstellt und direkt in die Autorenumgebung importiert werden. 3D-DREAMS fügt sich sehr gut in die Entwicklungsumgebung ein und bietet viele Funktionen zur Manipulation der Modelle mittels LINGO. Zur Zeit fehlt es der Zusatzsoftware an Stabilität und Verbesserung der Importfunktionalität. Da die Software frei erhältlich ist, gibt es keinen Anspruch auf Support. Nachteilig ist die Ungewissheit der Weiterentwicklung des Xtras. Nach aktuellen Informationen entwickelt MACROMEDIA mit anderen Firmen eine 3D-Technologie für die Darstellung im Web via SHOCKWAVE. SHELLS INTERACTIVE ist an dieser Entwicklung beteiligt.

Eine Alternative stellt die Verwendung von JAVA mit der Erweiterung der JAVA3D-API dar. Sie hat den Vorteil einer zukunftsträchtigen und sehr flexiblen Technologie. Das GENLAB-Framework kann integriert werden. Zur Zeit gibt es noch einige, leider für die Erstellung virtueller Labore schwerwiegende Nachteile. Das sind v. a. das Fehlen einer geeigneten Importmöglichkeit professioneller 3D-Grafikformate, sowie der Unterstützung in der Erstellung multimedialer Elemente durch Multimedia-Designer mittels visueller Techniken.

Als Ergebnis dieser Analyse wird festgestellt, dass die Kombination des populären Autorensystems Macromedia DIRECTOR mit Shells Interactive 3D-DREAMS zur Integration dreidimensionaler Objekte den Anforderungen an die Entwicklung dreidimensionaler virtueller Labore am ehesten gerecht wird.

3.4 Zusammenfassung

Zusammenfassend wurde in diesem Kapitel festgestellt, dass geeignete Werkzeuge zur Erstellung virtueller dreidimensionaler Labore sowohl den Anforderungen der allgemeinen Multimedia-Softwareentwicklung, der Programmierung komplexer Abläufe und insbesondere der Integration dreidimensionaler Modelle gerecht werden müssen. Anhand eines detaillierten Kriterienkataloges wurden zwanzig Entwicklungswerkzeuge untersucht. Die Analyse zeigte, dass eine Kombination mehrerer Werkzeuge notwendig ist, um die breitgefächerten Aufgaben zu erfüllen. Die endgültige Wahl fiel auf

den Einsatz von Macromedia DIRECTOR mit der 3D-Erweiterung 3D-DREAMS von Shells Interactive.

4 Entwurf der 3D-Visualisierung

In diesem Kapitel werden Konzepte zur Integration dreidimensionaler Modelle in virtuelle Labore vorgestellt. Wie Kapitel 3 zeigte, ist die Vielfalt der auf dem Markt existierenden Entwicklungswerkzeuge für multimediale Anwendungen groß. Die Entwicklung interaktiver dreidimensionaler Welten wird erst langsam durch die stetige und schnelle Weiterentwicklung der Hardware vorangetrieben. Dies führt zur Zeit zu ständig und schnell ändernden Angeboten der Entwicklungswerkzeuge für interaktive 3D-Szenarien und deren Leistungsumfang. Die hier vorgestellten Konzepte berücksichtigen deswegen eine möglichst große Unabhängigkeit von den verwendeten Entwicklungswerkzeugen.

Der Entwurf gliedert sich in zwei Teile. Im ersten Teil wird die Architektur unabhängig von einer Entwicklungsumgebung entworfen. Dies gewährleistet eine leichte Übertragung der Konzepte auf beliebige objektorientierte Entwicklungsumgebungen. Im zweiten Teil (Abschnitt 4.2) wird der Entwurf in Anlehnung der in Kapitel 3 vorgeschlagenen Entwicklungsumgebung verfeinert, so dass eine Integration des 3D-Visualisierungskonzeptes in das GENLAB-Framework durchgeführt werden kann. Zum vollständigen Verständnis des zweiten Teils werden Grundlagen des GENLAB-Frameworks als bekannt vorausgesetzt.

Zur Darstellung und Verfeinerung der hier vorgestellten Konzepte werden Klassendiagramme gemäß der UML-Spezifikation [BRJ99] eingesetzt. Sie beschränken sich aus Gründen der Übersichtlichkeit auf die jeweils dem Kontext entsprechenden wichtigen Bestandteile.

4.1 Entwurf: Teil I

Dieser erste Teil des Entwurfes behandelt die allgemeine Architektur der 3D-Visualisierung anhand von eigenständigen Komponenten und deren Kommunikation untereinander. Es folgt eine Identifikation derjenigen Komponenten, die leicht von einander kapselbar sind und möglichst einfach austauschbar gehalten werden müssen. Insbesondere die Unabhängigkeit von dem „Kern“ des virtuellen Labors wird berückichtigt. Unter dem „Kern“ werden hier die Bestandteile verstanden, die unabhängig von der tatsächlichen Darstellung auf dem Bildschirm sind, beispielsweise Zustände von Laborkomponenten, Versuchsabläufe und Modellierung von visuellen Zusammenhängen (Views). In Abschnitt 4.1.2 werden die notwendigen Schnittstellen zur Realisierung der Kommunikation zwischen den Komponenten entworfen. Jeglicher Austausch von Daten oder

Kontrolle findet über diese Schnittstellen statt. Damit wird gewährleistet, dass bei Austausch einer Komponente nur diese zentralen Schnittstellenklassen angepasst werden müssen.

4.1.1 Identifikation der Komponenten

Die Anforderungen an eine 3D-Visualisierung für virtuelle Labore wurden in Kapitel 2 erörtert. Sie lassen sich bzgl. der *Visualisierung* in zwei Kategorien einteilen:

① Vom Kontext eines Versuches oder Labors *abhängige* Anforderungen an ein 3D-Objekt. Dazu gehören beispielsweise, Einschränkungen von Rotationen und Bewegungen, Änderung der Objekthierarchie aufgrund von Aktionen wie Elemente in einen Behälter einfügen, Drag & Drop-Aktionen, Ausführung von Prozessen aufgrund von Benutzerinteraktion usw.

② Vom Kontext eines Veruches oder Labors *unabhängige* Anforderungen. Allgemein zählen dazu die Verwaltung und die Darstellung der Szene, das Importieren und Verwalten von Objektgeometrien, -erscheinungsbild und -hierarchien, die Berechnung von Transformationen (Rotation, Skalierung, Translation), die Berechnung der eigentlichen Darstellung der Szene aus einer bestimmten Perspektive, die Erkennung von Kollisionen etc.

Der Entwurf läßt sich also in die zwei Komponenten „Kernlabor“, welche die kontextabhängigen Aufgaben übernimmt, sowie „Render-Engine“, welche die 3D-Szene verwaltet und darstellt, gliedern. Diese Trennung ist sinnvoll, da so die grafische Darstellung von der internen Logik getrennt ist. Sie wird in vielen „grafik-orientierten“ Anwendungen vorgenommen. Dazu gehören beispielsweise 3D-Spiele, Simulationsapplikationen, Webbrowser etc. Ein Grund dafür ist die Möglichkeit, eine Render-Engine in unterschiedlichen Applikationen einsetzen zu können. Auf dem Markt werden freie und kommerzielle Render-Engines angeboten. Ein Framework für virtuelle Labore mit 3D-Untersützung sollte daher die Verwendung einer speziellen Render-Engine nicht vorschreiben, sondern die Möglichkeit geben, sie ohne großen Aufwand auszutauschen. So kann beispielsweise zu verschiedenen Labortypen die jeweils geeigneteste Engine gewählt werden.

Die Kollisionserkennung ist für alle Systeme, in denen der Nutzer interaktiv mit 3D-Objekten arbeiten kann, ein wichtiger Bestandteil. Die meisten Render-Engines bieten daher oft eigene Kollisionserkennungsmechanismen an. Wie in Kapitel 2 gezeigt wurde, gibt es unterschiedlich genaue und unterschiedlich schnelle Kollisionserkennungsmethoden. Ob also die von der Render-Engine zur Verfügung gestellten Methoden geeignet sind oder ein anderer Kollisionsalgorithmus verwendet werden muss, hängt von dem konkreten Labor ab. Vorstellbar sind beispielsweise genaue (und dafür langsamere) Kollisionserkennungsmethode in Bereichen der mechanischen Physik, während in Chemielaboren eine weniger genaue Kollisionserkennung ausreichend sein kann. Dem Entwickler eines virtuellen Labors sollte die Möglichkeit frei gestellt werden,

selbst und unabhänging von der Render-Engine die Art der Kollisionserkennung zu wählen. Damit diese Wahl nicht die Entwicklung des Laborkerns beeinträchtigt wird in diesem Visualisierungskonzept die Kollisionsentdeckung als eigenständige Komponente betrachtet.

Der Entwurf wird also in Komponenten „Laborkern“, „Render-Engine“ und „Kollisionsentdeckung“ unterteilt. Es müssen geeignete Maßnahmen durchgeführt werden, um den Kontroll- und Datenfluss zwischen den Komponenten zu regeln. Abbildung 4.1 veranschaulicht die Zusammenhänge.

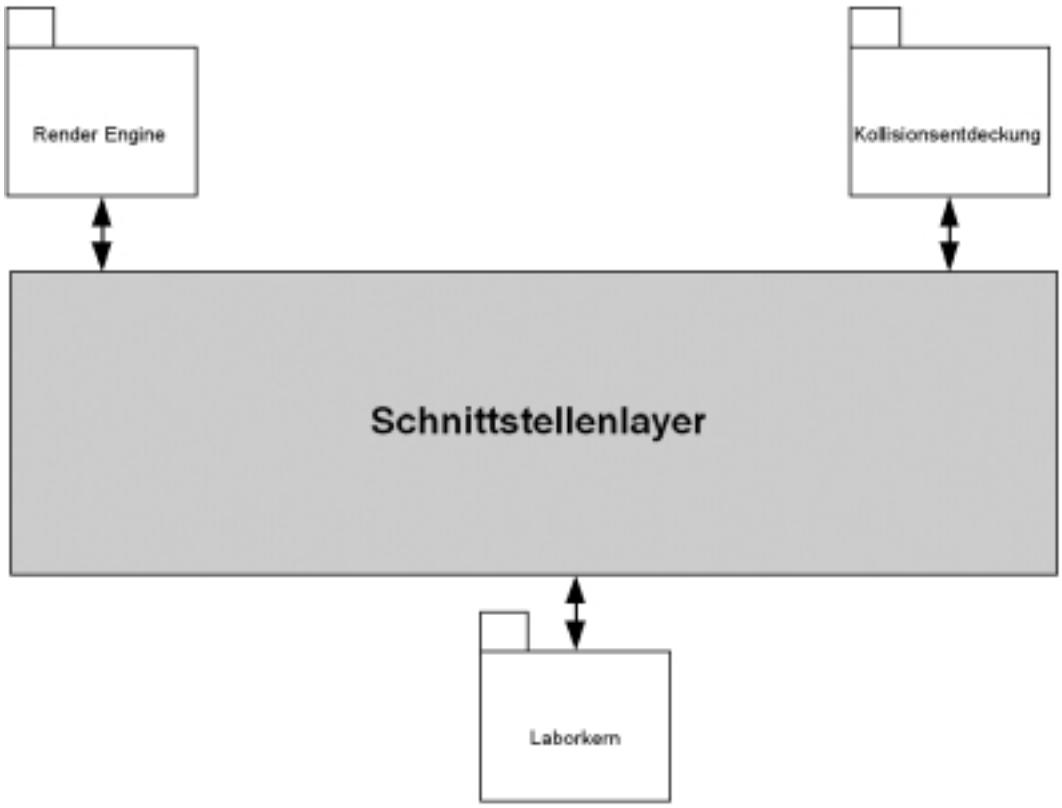

Abbildung 4.1: Die Komponenten für die 3D-Visualisierung eines virtuellen Labors

4.1.2 Schnittstellenlayer

Der Austausch einer Komponente durch eine andere soll möglichst ohne Änderung an einer anderen Komponente bzw. durch Änderungen an einer zentralen Stelle durchgeführt werden können. Zu diesem Zweck werden Klassen entworfen, innerhalb derer alle Zugriffe zwischen Laborkern, Render-Engine und Kollisionserkennung gekapselt werden. Alle Klassen, die Zugriff auf eine andere Komponente haben müssen, greifen nur über diese Schnittstellen auf die entsprechende Komponente zu.

Charakterisierung der 3D-Objekte

Jedes 3D-Objekt benötigt nur Zugriff auf bestimmte Methoden einer anderen Komponente. Beispielsweise braucht eine virtuelle Kamera keine Methode, um Geometrieeigenschaften zu ändern und eine Laborkomponente benötigt keine Methode zum Einstellen der Brennweite. Im folgenden werden die verschiedenen Objekte einer 3D-Szene identifiziert, auf die von mehreren Komponenten zugegriffen werden muss und deshalb in den Schnittstellenlayer aufgenommen werden müssen. Als Grundlage für diese Identifikation dient das Datei- und Scene Graph-Konzept von VRML, das mittlerweile zum ISO-Standard zur Beschreibung virtueller 3D-Welten geworden ist. Da nach [KRSD98] die Verbreitung von JAVA 3D-Szenen mittlerweile ebenso vorangeschritten ist wie VRML-Welten, wurden die Konzepte der Szenenverwaltung in JAVA 3D mitbetrachtet. Beide Konzepte basieren auf einer hierarchischen Gliederung der Objekte in Form einer Baumstruktur. Die Struktur einer Szene in VRML wird in Abbildung 4.2 verdeutlicht.

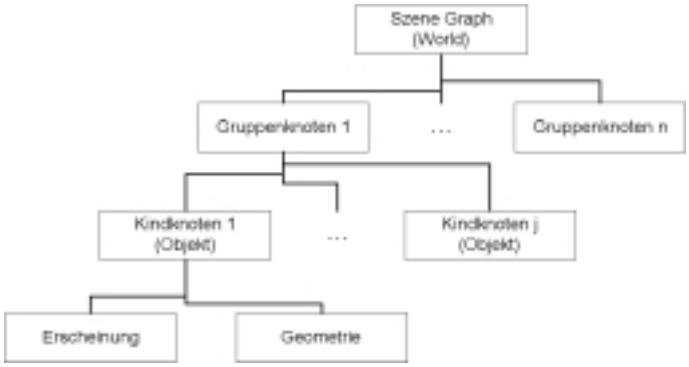

Abbildung 4.2: Hierarchische Objektstruktur einer VRML-Szene

In VRML beinhaltet eine Datei genau einen Szenen-Graphen. Dieser setzt sich aus beliebig vielen Gruppenknoten zusammen, die wiederum entweder aus weiteren Gruppenknoten oder aus Kindknoten bestehen. Ein Kindknoten setzt sich aus einer Geometrie und einem Erscheinungsbild (Material) zusammen. Jede Manipulation, beispielsweise Rotation oder Translation eines Gruppenknotens wird an seine Gruppen- und Kindknoten weitervererbt. *Diese Vererbung hat nichts mit der Vererbung aus programmiertechnischer Sicht zu tun. Eine Szene wird aus programmiertechnischer Sicht über Aggregation und Komposition und nicht über den Vererbungsmechanismus aufgebaut. Es werden lediglich die Manipulationen eines Objektes an alle seine Bestandteile weitergegeben.* In [Bußß99] wird die Szenenstruktur von JAVA3D beschrieben. Sie gleicht der VRML-Gliederung, unterscheidet sich aber in einer weiteren „Oberklasse“, dem *Universum*. Das Universum kann mehrere Szenen beinhalten, wobei jedoch immer nur eine Szene gleichzeitig angezeigt wird. Es verwaltet außerdem die Einstellungen der Größe und Positionen des Viewports. In VRML entspricht dem JAVA3D-Universum das Objekt, welches die Szene bzw. den VRML-Viewer einbettet, üblicherweise die Webseite. Soll eine Szene gewechselt werden, muss in VRML eine weitere Szenendatei geöffnet werden. Eine Szene in virtuellen Laboren kann mit einem Raum gleich-

gesetzt werden. Es kann immer nur ein Raum gleichzeitig betrachtet werden. Ein Raumwechsel und damit ein Szenenwechsel kann während der Experimentierphase häufig stattfinden. Da das Öffnen und Laden einer Szene viel Zeit in Anspruch nimmt, ist eine Objektstruktur wie die in JAVA 3D zu bevorzugen. In diesem Fall werden alle Objekte beim Starten geladen und die Hierarchie entsprechend aufgebaut, jedoch nur die in der aktiven Szene befindlichen Objekte angezeigt. Beim Wechsel eines Raumes muss nur spezifiziert werden, welche Szene beim Rendern berücksichtigt werden soll.

Gruppenknoten werden in JAVA 3D Transformationsknoten genannt. Nach diesem Szenenaufbau können bisher die Objekte *Universum, Szene* bzw. *Welt* und *Transformationsknoten* identifiziert werden. Sie sind lediglich „Hilfsobjekte“ und sind bei der Darstellung der Szene für den Betrachter nicht sichtbar. Folgende Objektarten werden in [KRSD98] beschrieben, um die Szene zu gestalten: Viewpoints / Kameras, Lichtquellen in verschiedenen Typen (Punktlicht, Spotlicht etc.), und verschiedene „Meshobjekte“, also die eigentlichen dreidimensionalen Objekte, die in einer Szene zu sehen sind. In virtuellen Laboren sind Lichtquellen eher uninteressant. Sie können zwar den Realitätseindruck erhöhen, tragen aber keinen weiteren Nutzen bei der Interaktion in virtuellen Laboren bei. Kameras können hingegen zur Navigation innerhalb der Laborumgebung genutzt werden (2.3.5). „Meshobjekte“ sind elementar für die Darstellung von Laborkomponenten.

Insgesamt konnten folgende für virtuelle Labore relevante Objektarten identifiziert werden:

- ☞ Universum
- ☞ Welt
- ☞ Transformationsknoten
- ☞ Kamera
- ☞ echte 3D-Objekte / Meshobjekte

Für diesen Entwurf wird vorausgesetzt, dass die Render-Engine diese oder äquivalente Klassen zur Verfügung stellt. Diese Voraussetzung darf angenommen werden, da diese Objektarten eine Untermenge des ISO-Standards zur Beschreibung virtueller Welten darstellen.

Einfluss des Vererbungsmechanismus auf die Gestaltung der Schnittsellenklassen

Teilweise ist eine Hierarchisierung der Schnittsellenklassen sinnvoll. Dies ist beispielsweise bei der Positionsroutine eines Objektes der Fall. Sowohl eine Kamera als auch ein Meshobjekt kann die Positionsroutine einer Oberklasse nutzen. Dieses Konzept kann jedoch nur in Sprachen, die Mehrfachvererbung unterstützen, implementiert werden. Abbildung 4.3 veranschaulicht diese Zusammenhänge:

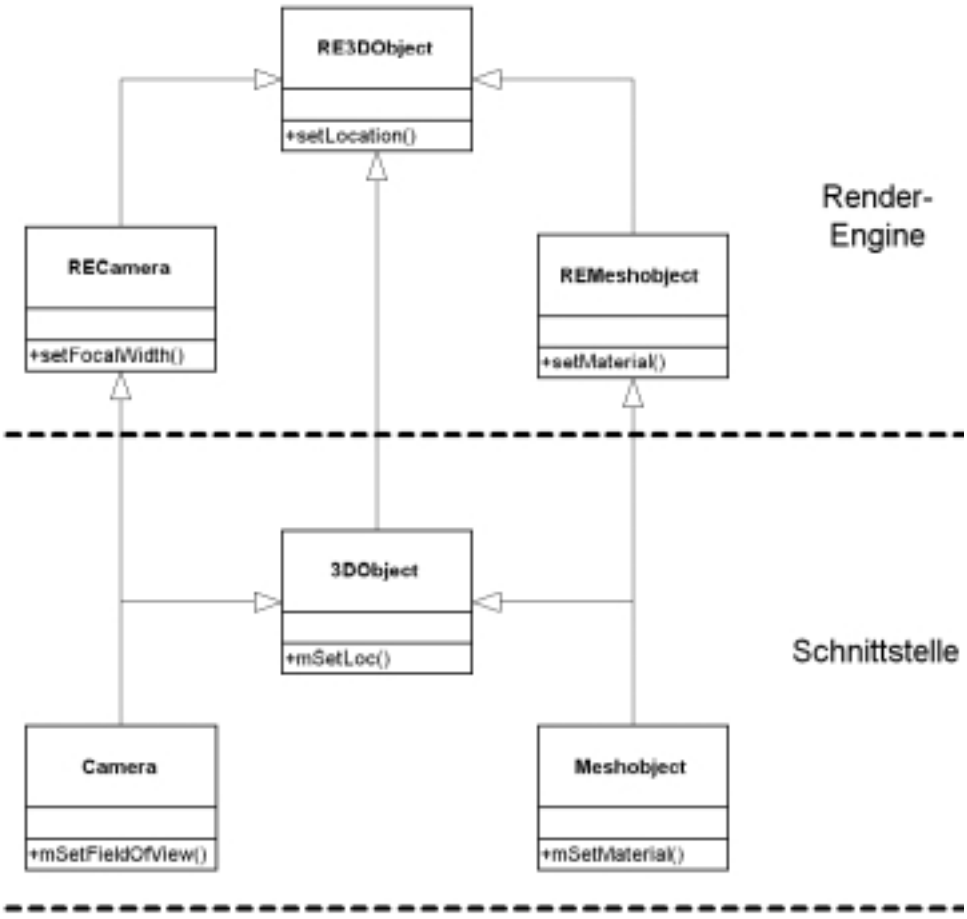

Abbildung 4.3: Hierarchische Schnittstelle mit Mehrfachvererbung

Die Positionsroutine der Schnittstelle kann in einer Klasse `3DObject` implementiert werden und von einer entsprechenden Klasse der Render-Engine (hier `RE3DObject`) abgeleitet werden. Die Klasse `Camera` sowie die Klasse `Meshobject` können diese Routine nutzen. Diese beiden Schnittstellenklassen müssen jedoch auch auf ihre „spezifischen“ Methoden zugreifen, beispielsweise die Einstellung der Brennweite beim Kameraobjekt oder das Setzen des Materials beim Meshobjekt. Dazu können diese Klassen neben einer Ableitung von der Klasse `3DObject` von den jeweiligen Klassen der Render-Engine abgeleitet werden (im Beispiel von der Klasse `RECamera` bzw. `REMeshobject`). Ohne Mehrfachvererbung muss die Positionsroutine `mSetLoc()` jeweils in den einzelnen Klassen dupliziert werden (s. Abbildung 4.4). Ein gemeinsame Oberklasse, in der die Positionsroutine implementiert wird, kann nicht verwendet werden.

Der Mehrfachvererbungsmechanismus ist nicht in allen objektorientierten Sprachen vorhanden. Deswegen nutzt dieses Visualisierungskonzept nur die Einfachvererbung aus.

Innherhalb des Schnittstellenlayers kann es damit keine Vererbungshierarchie, denn jede Klasse hat als einzigen Vorfahren eine äquivalente Klasse aus der Render-Engine.

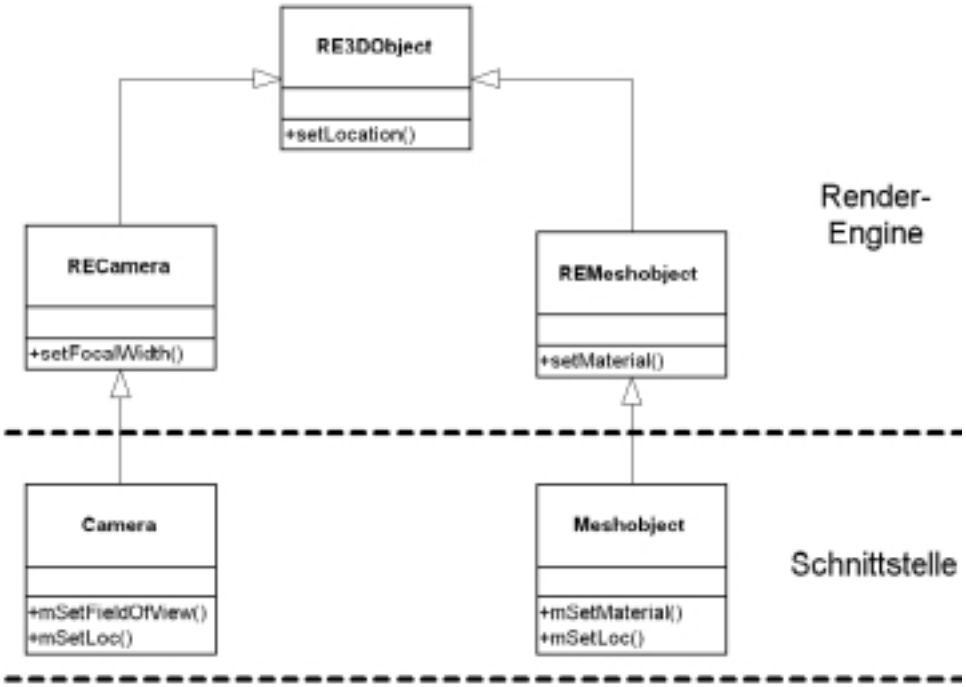

Abbildung 4.4: Bei Einfachvererbung können keine Routinen in eine Oberklasse ausgelagert werden

Da die für virtuelle Labore wichtigen 3D-Objekte bereits genannt wurden, kann der Schnittstellenlayer wie in Abbildung 4.5 dargestellt werden:

Die folgenden Unterkapitel beschreiben die in den einzelnen Schnittstellenklassen zu implementierenden Zugriffsmethoden.

4.1.3 Schnittstellenklasse `3DView`

Die Schnittstelle `3D-View` beinhaltet alle Methoden, die zum Zugriff auf Meshobjekte von verschiedenen Komponenten aus benötigt werden.

Laborkern / Render-Engine

Wie in Kapitel 2 gezeigt wurde, besitzen Meshobjekte viele während der Laufzeit einstellbare Eigenschaften. Sie werden durch den Laborkern kontrolliert und manipuliert, aber letztendlich an die Render-Engine weitergereicht. Folgende Methoden werden in diese Schnittstelle aufgenommen:

① Erzeugung und Löschung des Objektes, Initialisierung etc.

☞ `mNew():meshobject`
Erzeugung eines neuen 3D-Objektes

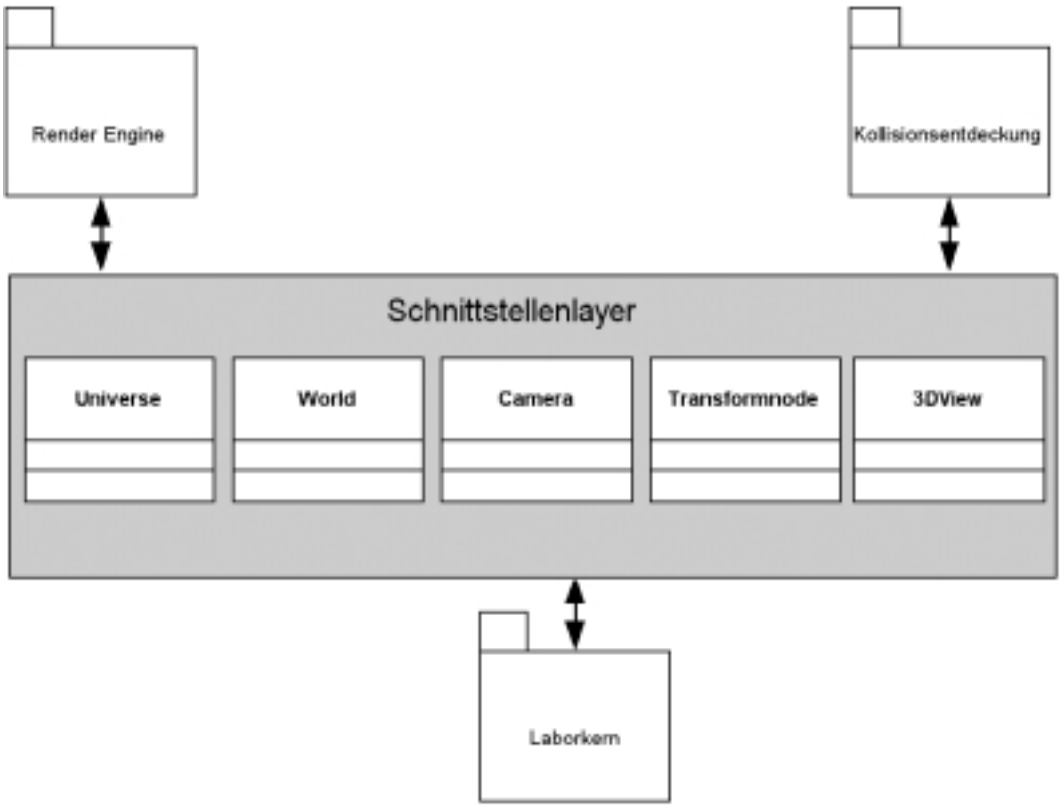

Abbildung 4.5: Darstellung des Schnittstellenlayers

- ☞ `mDestruct()`
  Entfernen eines Objektes aus der 3D-Szene
- ☞ `mCreateTransformnode():transformnode`
  Erzeugung eines neuen Gruppenknoten bzw. Transformationsknotens

② Manipulation der Objekthierarchie

- ☞ `mSetParent(Object)`
  Anhängen eines 3D-Objektes oder Transformationsknoten an einen Vater (bzgl. des hierarchischen Aufbaus der 3D-Szene)
- ☞ `mGetParent():Object`
  Liefert den Vater eines 3D-Objektes zurück.
- ☞ `mGetChildren():List of Object`
  Liefert eine Liste der Objekte, die unterhalb des Objektes angesiedelt sind.
- ☞ `mGetWorld():world`
  Gibt die Welt/Szene zurück, in der sich das Objekt befindet.

③ Transformationen

In der Problemanalyse (Kapitel 2) wurden drei für virtuelle Labore notwendige Transformationsarten festgestellt:

a) Translation (Positionierung)

b) Rotation und optional

c) Skalierung

Bei der Bestimmung von Lage und Orientierung im Raum werden nach [Heu99] üblicherweise entweder ein Objektkoordinatensystem oder das globale Koordinatensystem verwendet. Bei der Angabe des globalen Koordinatensystems beziehen sich alle Positionierungs- und Orientierungskoordinaten auf den Raum, unabhängig von anderen Objekten. Die Angabe eines Objektkoordinatensystems transformiert das Objekt relativ zu diesem Koordinatensystem. Das Referenzobjekt wird den Transformationsmethoden jeweils als Parameter übergeben. Besitzt das Objekt einen Transformationsknoten, wird das Objekt nicht direkt sondern nur indirekt über diesen Transformationsknoten transformiert. Dies ermöglicht beispielsweise eine Rotation eines Objektes nicht um die eigene Objektachse, sondern um die Achse des Transformationknoten (wichtig bei der Rotation von Türen). Folgende Methoden werden zur Transformation von Objekten in die `3DView`-Klasse aufgenommen:

- ☞ `mSetLoc(location, refobject)`
  Diese Methode positioniert das Objekt unter Angabe eines Koordinatensystems (eigenes, globales oder das eines anderen Objektes) relativ zu diesem Koordinatensystem. Die Position wird in dreidimensionalen Welten üblicherweise nicht in Pixel, sondern in Metern, Zentimetern oder Inches angegeben.
- ☞ `mGetLoc(refobject):location`
  Liefert unter Angabe eines Koordinatensystems die von diesem Koordinatensystem relative Position zurück.
- ☞ `mSetOrientation(orient, refobject)`
  Setzt die Orientierung eines Objektes abhängig von einem angegebenen Koordinatensystem. Die Orientierung wird üblicherweise in Winkel angegeben. Im Gegensatz zur Positionierung werden die Winkel nicht bezüglich des (X,Y,Z)-Koordinatensystems, sondern bezüglich des (Yaw,Pitch,Roll)-Koordinatensystems angegeben. Der Vorteil dieses Koordinatensystems liegt in der Entkopplung der Winkel. Die Angabe der Reihenfolge der Objektdrehung um einen Yaw, Pitch oder/und Roll-Wert ist irrelevant. Ob zuerst um Yaw oder Pitch gedreht wird oder umgekehrt, liefert dieselbe Orientierung im Raum.
- ☞ `mGetOrientation(refobject):orient`
  Gibt die Orientierung eines Objekts bzgl. eines angegebenen Koordinatensystems zurück.
- ☞ `mSetSize(factor)`
  Skaliert ein Objekt auf die Größe `factor`. Der Faktor enthält drei Werte (x,y,z), die jeweils die absolute Dimension der angeben.
- ☞ `mGetSize():faktor`
  Liefert die absolute Größe des Objekts zurück.

④ Verwaltung von Objekteigenschaften (Geometrie, Material)

In Kapitel 2 wurden die Eingeschaften dreidimensionaler Objekte beschrieben. Sie lassen sich in zwei Kategorien einteilen:

a) Geometrie

b) Erscheinung (Material u. Texturen)

Für die Manipulation von Objekten kann es unter Umständen notwendig sein, diese Eigenschaften auch während der Laufzeit zu verändern. Geometrieänderungen können beispielsweise bei der Darstellung von verschiedenen Füllständen eines Behälters eingesetzt werden. So müssen nicht mehrere 3D-Objekte zur Repräsentation verschiedener Füllstände herangezogen werden. Die dynamische Änderung der Erscheinungseigenschaften hat mehrere Einsatzgebiete. Beispielsweise kann das Aussehen einer Flüssigkeit verändert werden, digitale dynamisch ändernde Anzeigen (z. B. Zeitanzeige der Mikrowelle) können dargestellt werden, und sie kann zur Hervorhebung von Objekten bei einer Interaktion verwendet werden. Folgende Methoden werden deshalb in die Klasse `3DView` aufgenommen:

- ☞ `mSetGeometry(geometry)`
  Setzt die geometrische Struktur eines 3D-Objektes.
- ☞ `mGetGeometry():geometry`
  Liefert die geometrische Struktur eines 3D-Objektes zurück.
- ☞ `mSetMaterial(material)`
  Setzt die Materialeigenschaften eines 3D-Objektes.
- ☞ `mGetMaterial():material`
  Liefert die Materialzusammensetzung eines Objektes.
- ☞ `mSetTexture(texture)`
  Setzt eine Textur auf ein 3D-Objekt.
- ☞ `mGetTexture():texture`
  Gibt die Textur eines 3D-Objektes zurück.
- ☞ `mImportGeometry(file, name):geometry`
  Importiert die Geometrie `name` aus der Datei `file`.
- ☞ `mImportMaterial(file, name):material`
  Importiert das Material `name` aus der Datei `file`.
- ☞ `mImportTexture(file):texture`
  Importiert die Datei `file` als Textur.

⑤ Handling von Maus- und Tastaturereignissen

Auch auf dem umgekehrten Wege, d. h. von der Render-Engine zum Laborkern kann es zur Kommunikation kommen. Dies betrifft das Handling von Mausereignissen, die von der Render-Engine an das betreffende 3D-Objekt geschickt und dort weiterverarbeitet werden müssen.

Die Interaktion mit den 3D-Objekten basiert auf zweidimensionale Eingabegeräte, also Tastatur und 2D-Maus. Sie müssen auf die dreidimensionale Darstellung „gemapped“ werden. Das „Mappen“ übernimmt die Render-Engine. Es bedeutet, die Mauspositionen auf dem zweidimensionalen Viewport auf Positionen in den 3D-Raum abzubilden und das unter dem Mauszeiger befindliche Objekt zurückzuliefern. Die Eingabeinteraktionen sind durch die Eingabegeräte begrenzt. Diese sind dieselben wie in GENLAB. Deshalb werden hier die gleichen Methoden verwendet. Sie werden jeweils aufgerufen, wenn ein entsprechender Event auf einem 3D-Objekt ausgelöst wird:

- `mMouseDown()`
  Einfacher Klick der linken Maustaste auf das 3D-Objekt.
- `mMouseUp()`
  Die linke Maustaste wird auf diesem Objekt losgelassen. Das Objekt erhält jedoch nur diesen Event, wenn es auch den entsprechenden `mMouseDown`-Event bekommen hat.
- `mRightMouseDown()`
  analog zu `mMouseDown()`.
- `mRightMouseUp()`
  analog zu `mMouseUp()`.
- `mMouseEnter()`
  Der Mauszeiger tritt in den Bereich des Objektes ein.
- `mMouseLeave()`
  Der Mauszeiger verlässt den Bereich des Objektes
- `mMouseWithin()`
  Diese Methode wird aufgerufen, wenn sich der Mauszeiger auf dem Objekt befindet.
- `mMouseUpOutside()`
  Es erfolgte ein Mausklick auf dem Objekt, die Maustaste wurde jedoch außerhalb des Objektes wieder losgelassen.

Tastaturereignisse werden nur in Eingabefelder von Dialogboxen verwendet. Diese Arbeit berücksichtigt keine dreidimensionalen Dialogboxen, weshalb Tastaturereignisse nicht in die Schnittstellenklasse aufgenommen werden.

Laborkern / Kollisionserkennung

Damit das Kollisionshandling initialisiert und auf Kollisionen zweier Meshobjekte im Laborkern entsprechend reagiert werden kann, muss die Kommunikation zwischen Laborkern und Kollisionserkennung sichergestellt werden. Die Methoden für diese Kommunikation brauchen ausschliesslich in dieser Schnittstellenklasse (`3D-View`) implementiert werden. Die anderen 3D-Objekttypen sind virtuelle, also nur zu Szenenorganisation notwendige Objekte und können nicht kollidieren.

Die Kollisionserkennung arbeitet auf Basis des Szenengraphen. Bei einer Veränderung der Lage oder Orientierung eines Objektes wird überprüft, ob das sich bewegende Objekt ein anderes Objekt in der aktuellen Szene geschnitten hat. Um diesen aufwendigen Prozess zu beschleunigen (siehe Abschnitt 2.3.4), sollten nur die Objekte bei der Kollisionsverarbeitung geprüft werden, die für eine Kollision wirklich wichtig sind. Beispielsweise muss nicht jeder Schalter eines Gerätes überprüft werden, sondern meistens reicht eine Überprüfung des gesamten Gerätes auf eine Kollision. Das setzt voraus, dass es Methoden gibt, mit denen ein 3D-Objekt festlegen kann, ob es empfänglich für Kollisionen ist oder nicht. An Kollisionen sind jeweils zwei Objekte beteiligt. Weitere Methoden sind notwendig, um festzustellen ob seit der letzten Bewegung eine Kollision stattfand, und wenn ja, um das zweite an der Kollision beteiligte Objekt zu ermitteln. Hierfür sind zwei Methoden vorgesehen. Die Abfrage nach dem Stattfinden einer Kollision und die Abfrage mit welchem Objekt kollidiert wurde, werden getrennt behandelt. Der Grund dafür ist, dass es unter Umständen für einige Objekte irrelevant ist, mit welchem Objekt kollidiert wurde. Die Information, dass eine Kollision stattfand reicht aus. Die aufwendige Auswertung, welches das zweite an der Kollision beteiligte Objekt ist, kann dann entfallen.

Folgende Kollisionsmethoden werden daher in die Schnittstelle aufgenommen:

- `mSetSolidness(boolean)`
  Schaltet die Berücksichtigung bei der Kollisionserkennung ein oder aus.

- `mGetSolidness():boolean`
  Liefert zurück, ob das Objekt bei der Kollisionserkennung berücksichtigt wird.

- `mHasCollided():boolean`
  Weist die Kollisionsentdeckung an, zu überprüfen, ob dieses Objekt eine Kollision bei der letzten Transformationsänderung verursacht hat. Wird üblicherweise direkt nach einer Transformation (Bewegung, Rotation oder Skalierung) aufgerufen.

- `mLastCollisionObj():meshobject`
  Falls eine Kollision stattfand, kann mit dieser Methode das Kollisionsobjekt ermittelt werden.

4.1.4 Schnittstellenklasse `Camera`

Als „virtuelle“ 3D-Objekte ohne Geometrie und Materialeigenschaften können Kameraobjekte nicht kollidieren. Aus diesem Grund braucht die Kollisionserkennungskomponente keinen Zugriff auf Kameraobjekte und deshalb keinen Zugriff auf die Schnittstellenklasse `Camera`. Lediglich der Laborkern benötigt Zugriff auf die von der Render-Engine zur Verfügung gestellten Kamera(s).

Laborkern / Render-Engine

Kameras sind virtuelle Objekte, die die Render-Engine anweisen, die Szene aus einer bestimmten Perspektive mittels einer bestimmten Projektionsart in ein Fenster zu projizieren. Es gibt nur wenige Manipulationsmöglichkeiten, die für virtuelle Labore wichtig sind, nämlich neben den Erzeugungs- und Initialisierungsmethoden die Positions- und Orientierungsänderung. Diese reichen aus, um zwischen Ansichten zu wechseln und ggf. Kamerafahrten zu erstellen. Desweitern müssen Kameras in Szenen integriert werden, d. h. sie müssen an eine Welt oder ein anderes Objekt angehängt werden.

- ☞ `mNew():meshobject`
- ☞ `mDestruct()`
- ☞ `mCreateTransformnode():transformnode`
- ☞ `mInit()`

- ☞ `mSetParent(Object)`
- ☞ `mGetParent():Object`
- ☞ `mGetChildren():List of Object`
- ☞ `mGetWorld():world`

- ☞ `mSetLoc(location, refobject)`
- ☞ `mGetLoc(refobject):location`
- ☞ `mSetOrientation(orient, refobject)`
- ☞ `mGetOrientation(refobject):orient`

4.1.5 Schnittstellenklasse `Transformnode`

Ein Transformationsknoten ist ein virtuelles Objekt, das in eine Szene integriert werden kann. Es dient als Referenzobjekt bei Transformationen anderer Objekte. Wie die Kamera besitzt es lediglich eine Position, Orientierung und eine Einordnung in der 3D-Szene. Die Methoden sind also dieselben wie die der Klasse `Camera`. Eine spätere Erweiterung läßt sich einfacher durchführen, wenn die beiden Objekte dennoch getrennt behandelt werden.

4.1.6 Schnittstellenklasse `World`

Es werden nur diejenigen Objekte, welche sich bzgl. der Hierarchie der 3D-Szene unterhalb eines „Weltobjektes“ befinden, bei der Berechung und Darstellung dieser Welt berücksichtigt. Eine Welt wird dargestellt, wenn es innerhalb dieser Welt eine aktive Kamera gibt, d. h. wenn die Render-Engine diese Kamera zur Berechnung der Szene verwendet. Das Weltobjekt ist demnach nur ein Organisationsobjekt. Zugriffe von der Kollisionskomponente gibt es keine. Die Klasse `World` muss deshalb nur Methoden zum Zugriff von dem Laborkern auf die Render-Engine beinhalten:

Labokern / Render-Engine

- `mNew():meshobject`
- `mDestruct()`

Die Methode `mCreateTransformnode()` entfällt, da eine Welt nach VRML-Standard nicht transformiert werden kann.

- `mSetParent(Universe)`
- `mGetParent():Universe`
- `mGetChildren():List of Object`

4.1.7 Schnittstellenklasse `Universe`

Grundlegende Eigenschaften der Render-Engine werden über das Objekt `Universe` realisiert. Sie werden vom Laborkern aus manipuliert. Zugriffe von der Kollisionserkennung auf das Universum gibt es nicht.

Laborkern / Render-Engine

Vom Laborkern aus werden die Größe und Position des Viewports, die aktive Kamera und damit die darzustellende Welt festgelegt. Außerdem muss das Universum eine Funktion zur Verfügung stellen, die anhand der Mausposition, das Objekt zurückgibt, dass am nähesten der aktiven Kamera, also dem Nutzer, ist. Folgende Methoden werden in die Schnittstelle aufgenommen:

- `mNew():meshobject`
- `mDestruct()`
- `mGetChildren():List of Object`

- ☞ `mSetRect(rect)`
  Setzt die Größe und Position des Viewports gemäß `rect`.

- ☞ `mGetRect():rect`
  Gibt die Größe und Position des Viewports zurück.

- ☞ `mObjectAt(point):meshobject`
  Liefert das Objekt unter dem Mauszeiger zurück.

- ☞ `mSetActiveCamera(Camera)`
  Setzt die aktive Kamera, d. h. die Kamera, von der aus die Szene berechnet wird, auf `Camera`. Damit wird gleichzeitig die darzustellende Welt gewechselt, sofern sich die neue Kamera in einer anderen Welt befindet.

- ☞ `mGetActiveCamera():Camera`
  Liefert die aktive Kamera zurück.

Abbildung 4.6 veranschaulicht die Zugriffswege zwischen den Schnittstellenklassen (grau hinterlegt) und den Komponenten Laborkern, Render-Engine und Kollisionserkennung.

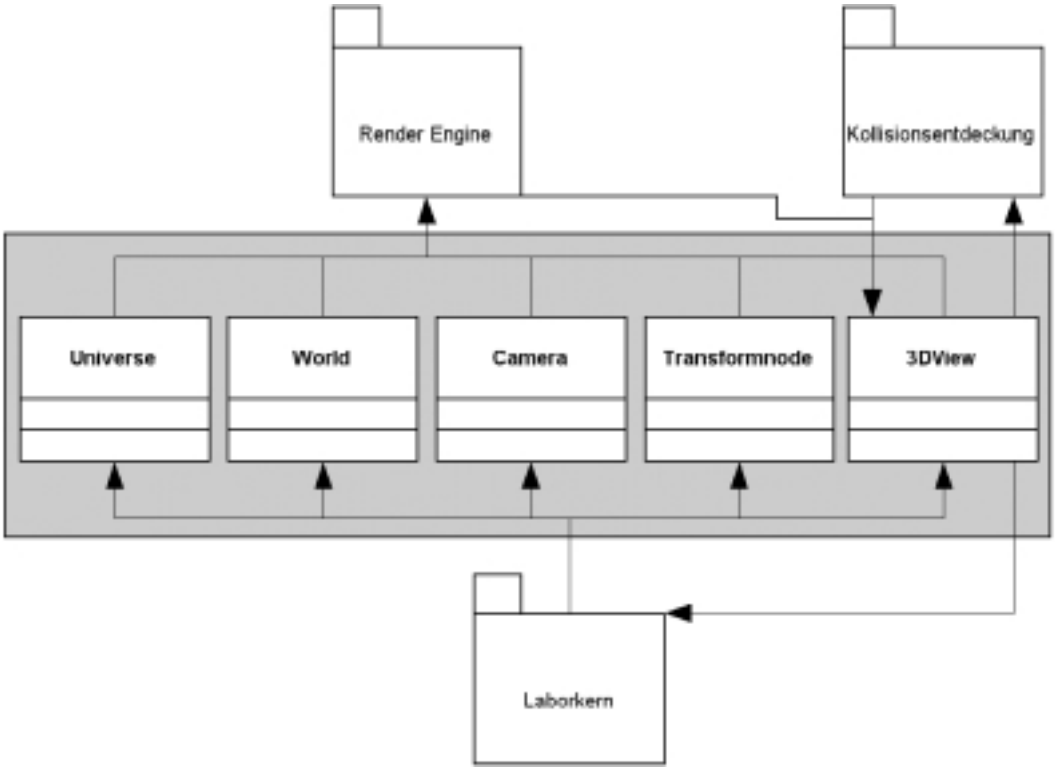

Abbildung 4.6: Schnittstellenlayer zwischen dem Laborkern, der Render-Engine und der Kollisionsentdeckung

4.2 Entwurf: Teil II

Der Feinentwurf des 3D-Visualisierungskonzeptes berücksichtigt die Fähigkeiten und Eigenschaften des Autorensystems MACROMEDIA DIRECTOR und des Xtras 3DDREAMS auf deren Basis die Implementation erfolgen soll. Ein Ziel des Entwurfes ist eine eine Anbindung der neuen „3D-Klassen“ an das GENLAB. Der Entwurf beinhaltet nicht das Design der Komponenten Render-Engine und Kollisionsentdeckung. Sie werden als „Black Boxes“ betrachtet.

Die Anbindung an das bzw. die Erweiterung des GENLAB-Frameworks umfasst mehrere Teilaufgaben, die in den folgenden Abschnitten erörtert werden sollen. Zur Einführung wird das GENLAB-Frameworks kurz beschrieben. Anschliessend folgt das Vorgehen zur Einbettung eines Viewports zur Darstellung dreidimensionaler Objekte in die Bühne von DIRECTOR unter Verwendung des GENLAB-Frameworks. In Abschnitt 4.2.3 wird analysiert, welche Klassen direkt vom GENLAB-Framework verwendet oder verfeinert werden können und welche Erweiterungen für die 3D-Visualisierung notwendig sind.

4.2.1 Aufbau des GENLAB-Frameworks

Das GENLAB-Framework ist eine in der objektorientierten Skriptsprache LINGO implementierte Zusammenstellung von Klassen, welche die Erstellung von naturwissenschaftlichen Laboren vereinfacht. Als grundlegendes Entwurfsmuster wurde das Model-View-Controller Konzept verwendet. Es wird ausführlich in der Diplomarbeit von Sigurd Elfreich [Elf99] erläutert und kann zusammenfassend wie folgt beschrieben werden:

Model-View-Controller (MVC) - Konzept

Beim *MVC*-Entwurf werden die Modelleigenschaften (Model) eines Objektes von der visuellen Darstellung (View) getrennt. Dies hat den Vorteil, dass ein Objekt mehrere Darstellungsformen gleichzeitig annehmen kann oder die Darstellungsform ausgetauscht werden können. Das Objekt selbst kennt seine (audio-) visuelle Erscheinung(en) nicht, wodurch Änderungen an den Views die Modelle unberührt lassen. Um die Konsistenz zwischen Models und Views aufrecht zu erhalten, müssen sich die Views bei den Modellen registrieren lassen. Sie erhalten dann bei einer Änderung einer Modelleigenschaft eine Nachricht und können dementsprechend reagieren. Die Controller bilden die eigentliche Schnittstelle zum Benutzer. Sie werden von einem View aggregiert und überwachen die Events der Eingabegeräte. Erfolgt ein Event wird vom jeweiligen Controller bestimmt welche Aktionen auszuführen sind. Diese Aktionen können entweder Modelleigenschaften ändern oder teilweise direkt den View manipulieren (dies gilt nur für Positions- und Orientierungsänderungen).

Das GENLAB-Framework

In DIRECTOR werden Anwenderereignisse wie Mausevents an den entsprechenden Sprite geschickt, auf dem das Ereigniss stattfand. Auf das Model-View-Controller Konzept übertragen, bedeutet dies, dass die Events von der visuellen Repräsentation (View) abgearbeitet wird. Daraus ergab sich für das GENLAB-Framework die Designentscheidung, die strikte Trennung von View und Controller aufzuheben. Die Controller nehmen dort ebenfalls eine (audio-) visuelle Erscheinung ein (siehe Abbildung 4.7), während die Views eine Komposition mehrerer Controller darstellt und Operationen auf den zusammengesetzten View zur Verfügung stellen, beispielsweise das Behandeln von anderen gedroppten Views.

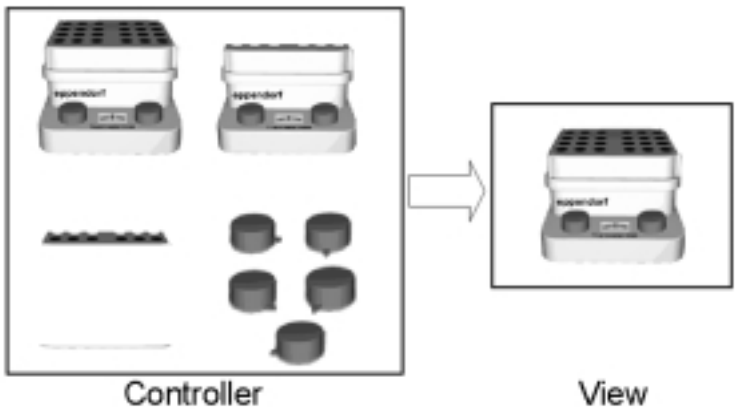

Abbildung 4.7: Komposition eines Views durch mehrere Controller im GENLAB-Framework

Vom GENLAB-Framework werden grundlegende Funktionalitäten in Form von abstrakten Model-, View- und Controllerklassen für die Erstellung konkreter (Labor-) Komponenten zur Verfügung gestellt. Ausserdem sind vorgefertigte Mechanismen zur Generierung von Versuchsaufbau, -abläufe und Laborverwaltung vorhanden.

Der grundlegende Aufbau des GENLAB-Frameworks wird in Abbildung 4.8 gezeigt.

Versuche werden in Module unterteilt. Hiefür stellt das GENLAB-Framework drei Varianten zur Verfügung:

① Standalone-Module: Sie stellen eigenständige Versuche dar und werden außerhalb eines anderen Versuch oder Teilversuchkontextes gestartet.

② Versuchs-Module: Ähnlich wie Standalone-Module, nur werden sie innerhalb eines Gesamtversuches gestartet. Laborkomponenten können zwischen den Modulen ausgetauscht werden.

③ Fertigkeiten-Module: Einfache Szenarien, um Anwendern eine Fertigkeit beizubringen, beispielsweise das Bedienen der virtuellen Pipette.

Abbildung 4.8: Die Basisklassen des GENLAB-Frameworks

Die Aufgaben der Module sind:

- Erzeugen der für das Module notwendigen Laborkomponenten
- Erzeugen der GUI-Komponenten (Anleitungsfenster, Nachrichtenfenster etc.)
- Erzeugen der Objekte, die den Versuchs-/Fertigkeitenablauf steuern
- Erzeugen der Arbeitsflächen
- Erzeugen der globalen Komponenten (z. B. des Labels)
- Positionierung der Laborkomponenten
- Positionierung der GUI-Komponenten
- Durchführung von für den Start des Moduls notwendigen Aktionen
- Darstellung aller Komponenten
- Reset und Löschen des Modules

Neue Versuche können erstellt werden, indem neue Module implementiert werden. Die neuen Module leiten sich von einer der drei oben beschriebenen Klassen ab.

Eine weitere wichtige Klasse stellt der `Labormanager` da. Er dient zum Starten und Wechseln von Modulen, sowie zum Zurücksetzen gestarteter Module. Weiterhin können von ihm aus die Arbeitsflächen eines Moduls gewechselt werden.

Außerdem bietet das GENLAB-Framework Klassen zur Einbindung von GUI-Komponenten und zur Kontrolle von Versuchsabläufen, die hier nicht weiter beschrieben werden.

4.2.2 Anbindung des Viewports / 3D DREAMS an das GENLAB-Framework

Durch Einbindung des Xtras 3D DREAMS lassen sich über die Menüauswahl oder per Lingo *Viewports* erstellen. Ein Viewport ist eine Art virtuelles Fenster innerhalb dessen die 3D-Szene gerendert wird. Nicht innerhalb dieses Fensters liegende 3D-Objekte werden bei der Bildberechnung nicht berücksichtigt. Welche Objekte sichtbar sind, wird einzig durch die aktive Kameraposition und Orientierung bestimmt und nicht durch die Größe des Viewports. Wird der Viewport verkleinert, werden alle Elemente der 3D-Szene im gleichen Verhältnis verkleinert. 3D DREAMS stellt die Möglichkeit zur Verfügung, mehrere Viewports in die Bühne einzubinden. Bei der Erstellung eines Viewports wird ein neuer Castmember angelegt. Castmember dienen in DIRECTOR dazu, verschiedene Medientypen zu speichern und zu organisieren. Über diesen neu erzeugten Castmember lassen sich 3D-Modelle importieren oder Primitive (einfache 3D-Körper) in die Szene integrieren, ihre Dimensionen und Erscheinungseigenschaften ändern. Die Szene des Viewports besteht standardmäßig aus einem Universum, mindestens einer Welt, einer Kamera und einer Lichtquelle. Mehrere Viewports sind völlig unabhängig, d. h. wird ein 3D-Modell in die Szene eines Viewports importiert, ist es in den Szenen anderer Viewports nicht vorhanden. Die „Viewport-Castmember“ können in Form von Sprites auf der Bühne platziert und skaliert werden. Die Größe eines Viewports beispielsweise kann über das `rect` des Sprites, in dem sich der Viewport befindet, verändert werden. Sollen andere Sprites hinter oder vor dem Viewport sichtbar sein, muss die Option „No Direct To Screen“ über den Eigenschaftsdialog des Viewport-Castmembers aktiviert werden. Das bedeutet zwar einen merkbaren Geschwindigkeitsverlust bei der Darstellung, jedoch können nur so die GUI-Elemente des GENLAB-Frameworks, die in anderen Spritekanälen platziert sind, direkt genutzt und über den Viewport positioniert werden.

Da ein 3D DREAMS -Viewport in DIRECTOR genauso behandelt werden kann wie eine Bitmap, kann er unter Verwendung des GENLAB-Frameworks als „gewöhnliches“ Viewobjekt in die Bühne eingebunden werden. Dazu wird eine neue Viewklasse geschaffen (`classViewUniverse`). Diese Klasse aggregiert einen neuen Controller `classControllerUniverse`, der wiederum direkt von der Klasse `classControllerBitmap` abgeleitet wird. Er übergibt seinen Vorfahren jedoch keine Bitmap, sondern den Castmember des 3D DREAMS -Viewports. Da der Viewport unter Umständen transparent sein kann, kann der Klasse `classViewUniverse` eine Liste von Hintergrundbitmaps übergeben werden, die hinter den Viewport gelegt werden können.

Von der Klasse `classViewUniverse` können spezialisierte Klassen abgeleitet werden. Diese können z. B. die Größe des Viewports, die Position des Viewport oder die Hintergrundbitmaps festlegen. So ist es möglich verschiedene Viewports beispielsweise für die Experimentierumgebung oder die Rahmenhandlung zu definieren.

Die Klasse `classViewUniverse` erzeugt außerdem eine Instanz der Klasse `classModelUniverse` als Model. Diese Klasse ist direkt von der Klasse `classModel` abgeleitet und hat nur die Funktion, den Model-Typ festzulegen. Hierfür wird der Typ

`#viewport` verwendet.

Der Controller (Instanz von `classControllerUniverse`), der den Viewport als Bitmap auf die Bühne platziert, erhält alle möglichen auf ihm ausgeübten Events (Mouseevents). An dieser Stelle müssen diese Events an die entsprechenden 3D-Objekte der Szene weitergeleitet werden. Klickt der Nutzer beispielsweise mit der linken Maustaste auf die 3D-Pipette, erhält der Controller des Viewports einen `mMouseDown()`-Event und noch nicht die Pipette. Der Controller muss nun diesen Event weiterleiten. Dazu erfragt er über die Schnittstellenmethode des Universums `mObjectAt` unter Angabe der aktuellen Mausposition, welches Objekt sich unterhalb des Mauszeigers befindet. An dieses Objekt wird der Event weitergeleitet. Diese Weiterleitung muss für alle Events, die der Viewport durch die Controllerklassen des GENLAB-Frameworks erhält, durchgeführt werden. Deshalb werden folgende Methoden in die Klasse `classControllerUniverse` implementiert:

- `mMouseWithin()`
- `mMouseDown()`
- `mMouseUp()`
- `mMouseRightUp()`
- `mMouseRightDown()`
- `mMouseEnter()`
- `mMouseLeave()`
- `mMouseUpOutside()`

Durch diese Weiterleitung wird sichergestellt, dass ein 3D-Objekt die gleichen Events wie eine normales 2D-Objekt erhält.

4.2.3 Anpassen und Erweitern des GENLAB-Frameworks für die 3D-Visualisierung

Zur Integration dreidimensionaler Objekte in virtuelle Labore, bedarf es, einige Klassen des GENLAB-Frameworks zu erweitern. Alle Klassen, die in keiner Weise in Verbindung mit der Visualisierung stehen, können ungeändert auch mit der 3D-Integration genutzt werden.

Models

Bei der 3D-Visualisierung der Laborkomponenten können die Models, also die Klassen, die nur interne Zustände speichern und Operationen auf ihren internen Zuständen ausführen, ohne jegliche Anpassung übernommen werden. Gerade die Trennung der Modelleigenschaften von der Repräsentation ist der Vorteil des MVC-Entwurfsmuster.

Views

Die Viewklassen aus dem GENLAB-Framework sind für die 3D-Visualisierung nicht verwendbar, denn ihre Methoden und Attribute basieren auf den zweidimensionalen Raum. Deshalb sind für die 3D-Visualisierung neue Viewklassen zu erstellen.

Die von einem Labor unabhängigen Eigenschaften der Views, also die Darstellung der Objekte, die Verwaltung der Hierarchie etc., werden bereits in Klassen der Render-Engine implementiert. Diese Klassen müssen so verfeinert werden, dass versuchs- und laborkontextabhängige Vorgänge (Drag & Drop, Behandlung von Kollisionen usw.) realisiert werden können. Für die Spezialisierung wurden im ersten Teil des Entwurfes die Schnittstellenklassen entworfen. Die neuen 3D-Viewklassen werden von diesen Schnittstellen abgeleitet. Die Klasse `3DView` stellt die Schnittstelle für alle Meshobjeke dar. Direkt von dieser Klasse wird die Klasse `3DViewObjects` abgeleitet. Sie beinhaltet die elementaren Bestandteile für 3D-Objekte in virtuellen Laboren, die nicht bereits durch die Render-Engine abgedeckt wurden. Diese werden im folgenden kurz beschrieben:

3DViewObjects:

- ☞ Initialisierung
  Die meisten Objekte müssen nach der Erzeugung oder beim Rücksetzen eine Initialisierung durchführen. Sie wird von der GENLAB-Framework -Klasse `classModule` gleichermaßen für alle Viewobjekte aufgerufen, so dass jedes Viewobjekt eine Init-Routine definiert haben muss. Von der Klasse `classModule` werden jedoch nur die „Vaterobjekte“ der Laborkomponenten initialisiert, beispielsweise die Mikrowelle. Deren Childobjekte (z. B. Tür, Türknopf) werden nicht initialisiert. Deswegen muss es eine Methode geben, die diese Initialisierung an die Childobjekte weitergibt.
- ☞ Model/Controller Erzeugung
  Jeder View besitzt ein Modell und evtl. Controller. Sie werden gewöhnlich in den Blattklassen erzeugt und müssen in Attributen gespeichert werden.
- ☞ mUpdate-Routine
  Die Benachrichtigung eines Viewobjekts aufgrund einer Zustandsänderung eines Models findet über die `mUpdate()`-Routine statt. Sie wird deshalb in dieser Klasse als abstrakte Methode eingefügt.
- ☞ Anhängen an andere 3D-Objekte
  In virtuellen Laboren gibt es 3D-Objekte, die auf der einen Seite eigenständig, andererseits fest an andere Objekte gebunden sind. Dazu zählt beispielsweise die Pipettenspitze. Nicht aufgesteckt stellt sie ein eigenständiges Viewobjekt mit eigenem Controller dar. Wird sie aufgesteckt (an die Pipette angehängt), soll ihr eigener Controller nicht nur deaktiviert

werden, sondern alle Events der Spitze an das Vaterobjekt gesendet werden. Um solch eine Beziehung zu definieren, werden zwei Methoden zum Anhängen und Entfernen benötigt (`mAttachMe()` und `mDetachMe()`).

- ☞ Label / Tooltip
  Jedes Viewobjekt kann einen Label besitzen, der beim längeren Verharren des Mauszeigers auf diesem Objekt angezeigt wird. Der Labeltext kann dynamisch angepasst werden. Es werden daher Methoden zum Setzen und Auslesen des Labeltextes angeboten.
- ☞ Highlite / Unhighlite
  Standardmethoden zum Hervorheben und wieder Rücksetzen der Hervorhebung werden für jedes Viewobjekt angeboten.
- ☞ Methoden zur Behandlung von Material, Geometrie und Texturen
  Während der Laufzeit können Material- , Geometrie und Textureigenschaften verändert werden. Hierzu werden Methoden benötigt, die entsprechende Daten sichern und wieder zurückschreiben können.
- ☞ Animiertes Bewegen und Rotieren
  Die Schnittstelle zur Render-Engine bietet zwar Methoden zum sofortigen Ändern der Position und Orientierung, aber kein animiertes Bewegen und Rotieren. Diese Routinen werden in virtuellen Laboren zum Vorführen verwendet und werden deshalb in diese Klasse mit aufgenommen.

3DViewObjectsDrag:

Die Klasse `3DViewObjectsDrag` wird von der Klasse `3DViewOjects` abgeleitet und bietet Operationen zum Draggen und Droppen von Objekten, sowie vollständige Bearbeitung von Kollisionsereignissen. Es folgt eine genauere Beschreibung der beim Drag & Drop durchzuführenden Aktionen:

① Vorbereitende Maßnahmen (Methode `mDrag()`)

- ☞ Erfassung der notwendigen Informationen, um einen Rücksprung zu ermöglichen (`mGenerateDropBackInfo()`)
  Ein Rücksprung wird ausgeführt, falls ein illegaler Drop ausgelöst wurde. Zu den zu speichernden Informationen gehören die Position, Orientierung und das Vaterobjekt des zu bewegenden Objektes.
- ☞ Erstellung eines Drag-Bezugsrahmen (`mCreateDragReference()`)
  Das Objekt muss so bewegt werden, wie es der Anwender erwartet. Eine Mausbewegung beispielsweise nach links, soll das Objekt aus *seinem* Blickwinkel nach links bewegen. Als Basis des Dragbezugsrahmens wird daher die aktive Kamera gewählt. Da die aktive Kamera meistens etwas geneigt ist, weil auf einen Arbeitsplatz heruntergeguckt wird, würde das Objekt bei einer Verschiebung in die Tiefe (in z-Richtung) ebenfalls leicht nach unten bewegt werden. Aus diesem Grunde wird der Pitch-Winkel des Drag-Bezugsrahmen auf 0.0 Grad gesetzt.

- ☞ Aktualisierung der Szenen-Hierarchie
  Befindet sich das 3D-Objekt innerhalb eines Behälters, muss es beim Draggen aus diesem Behälter wieder entfernt werden und die Hierarchie entsprechend aktualisiert werden.

② Aufgaben während der Bewegung:

- ☞ Überprüfung ob eine Kollision stattgefunden hat (`mCheckCollisions()`)
  Während jeder Positionsänderung des Objektes, kann es mit anderen Objekten kollidieren. Bei jeder Positionsänderung wird deswegen die Routine `mCheckCollisions()` aufgerufen, die überprüft (bzw. durch die Kollisionsentdeckung überprüfen lässt), ob und mit welchem Objekt eine Kollision stattfand.
- ☞ Kollisionsverarbeitung
  Für den Fall einer Kollision müssen beide Kollisionpartner benachrichtigt und ggf. weitere Aktionen ausgeführt werden (z. B. Hilighting).
- ☞ Neupositionierung des Objektes (`mImproveLoc()`)
  Bei der Entdeckung einer Kollision darf das bewegte Objekt nicht an die vom Anwender verlangte Position bewegt werden, da sonst diese beiden Objekte ineinandertreten (überschneiden) würden. Stattdessen muss eine Position gefunden werden, bei der das Objekt seinen passiven Kollisionspartner nur berührt, jedoch nicht schneidet.

③ Drop des Objektes:

- ☞ Überprüfung auf eine Interaktion mit einem anderen Objekt
  Es muss geprüft werden, ob das gedroppte Objekt in Berührung eines anderen Objektes steht. Ist das der Fall, muss das passive Objekt benachrichtigt werden. Dieses führt evtl. weitere Aktionen durch.
- ☞ „Freier Fall“
  Wurde kein Interaktionspartner gefunden, wird das Objekt frei „fallengelassen“. Dabei können keine „echten“ Interaktionen mehr auftreten. Es kann jedoch sein, dass das Objekt auf ein anderes Objekt fällt. Dieses Objekt entscheidet wiederum, ob das fallengelassene Objekt auf diesem Objekt positioniert werden darf.
- ☞ Rücksprung `mDropBack()`
  Für den Fall, dass kein Interaktionsobjekt gefunden wurde und beim freien Fall das gedroppte Objekt auf ein Objekt fällt, das dieses nicht tragen kann, muss ein Rücksprung durchgeführt werden. Die Dropback-Information wird ausgelesen und der Rückspung durchgeführt.

Nach obiger Beschreibung gibt es zwei verschiedene Objektbeziehungen. Sie können entweder miteinander interagieren oder einfach nur getragen werden bzw. Träger sein. Jeder View kann für sich entscheiden, mit welchen Views er interagieren oder welche er tragen kann. Hierfür werden die beiden Methoden `mAddToDropTable()` und `mAddtoCarryTable()` zur Verfügung gestellt.

3DViewObjectsRepository:

Viele Objekte in virtuellen Laboren haben die Fähigkeit, andere Objekte aufzunehmen. Zu diesen Objekten gehören beispielsweise die Mikrowelle und das Eisgefäß. Zur Realisierung dieser Objektbeziehungen auf Viewebene besitzen sie eine „Einfügen“ und eine „Entfernen“-Methode. Diese Methoden werden in der Klasse `3DViewObjectsRepository` implementiert. Von dieser Klasse werden alle Behälter abgeleitet. Die mögliche Anzahl der einfügbaren Objekte wird durch das entsprechende Modell festgelegt. Die Einfügeoperation positioniert das einzufügende Objekt in diesem Visualisierungskonzept auf im konkreten View festgelegten Koordinaten und verändert die Objekt-Hierarchie, damit das eingefügte Objekt ein Childobjekt des Behälters wird. Wird ein Objekt innerhalb eines Behälters gedragged, wird die `mEntfernen()`-Methode aufgerufen.

Das Herausnehmen eines 3D-Objektes aus einem Behälter wurde in GENLAB realisiert, indem das herausgenomme Objekt durch Anpassung seiner Tiefenposition (z-Order) in den Vordergrund gestellt wird. Diese Entscheidung ist konsistent zu jedem anderen Drag in GENLAB, da ein gedraggtes Viewobjekt, auch wenn es sich nicht in einem Behälter befindet, in den Vordergrund rückt. In der 3D-Visualiserung behält das 3D-Objekt bei einem Drag die Tiefeninformation bei, solange es nicht explizit vom Anwender in der Tiefe verschoben wird. Eine Positionierung des 3D-Objektes in den Vordergrund würde den Anwender daher verwirren und ist in einer 3D-Visualisierung nicht geeignet. Deswegen wird analog zum Hineinstellen im konkreten View eine Position festgelegt, auf der das herausgedraggte Objekt platziert wird.

Bei Verwendung einer genauen Kollisionserkennung oder durch Verwendung zusätzlicher Objekte zur Kollisionserkennung könnte auch folgendermaßen vorgegangen werden. Der Nutzer muss das Objekt direkt in den Behälter draggen und dort loslassen. Eine genaue Kollisionserkennung erkennt die Polygone, auf denen das Objekt fallen gelassen wurde. So kann bestimmt werden, ob sich das Objekt auch wirklich im Inneren des Behälters befindet oder nur von außen berührt wurde.

Die bisher betrachteten Klassen dienen der Erstellung der sichtbaren Objekte. Sie werden übersichtshalber in der Abbildung 4.9 dargestellt.

Nach der Beschreibung aus Abschnitt 4.1 gibt es in einer 3D-Umgebung weitere Objektarten, die zwar keine visuelle Repräsentation besitzen und dadurch keine Controller aggregieren brauchen, aber durch Views modelliert werden können. Zu diesen Objekten gehören die Kameras, Transformationsknoten und Welten. Ihre Viewklassen werden im folgenden beschrieben:

Camera3D:

Die Klasse `Camera3D` wird von der Schnittselle `Camera` abgeleitet. Die für virtuelle Labore benötigten Operationen auf Kameras werden bereits von der

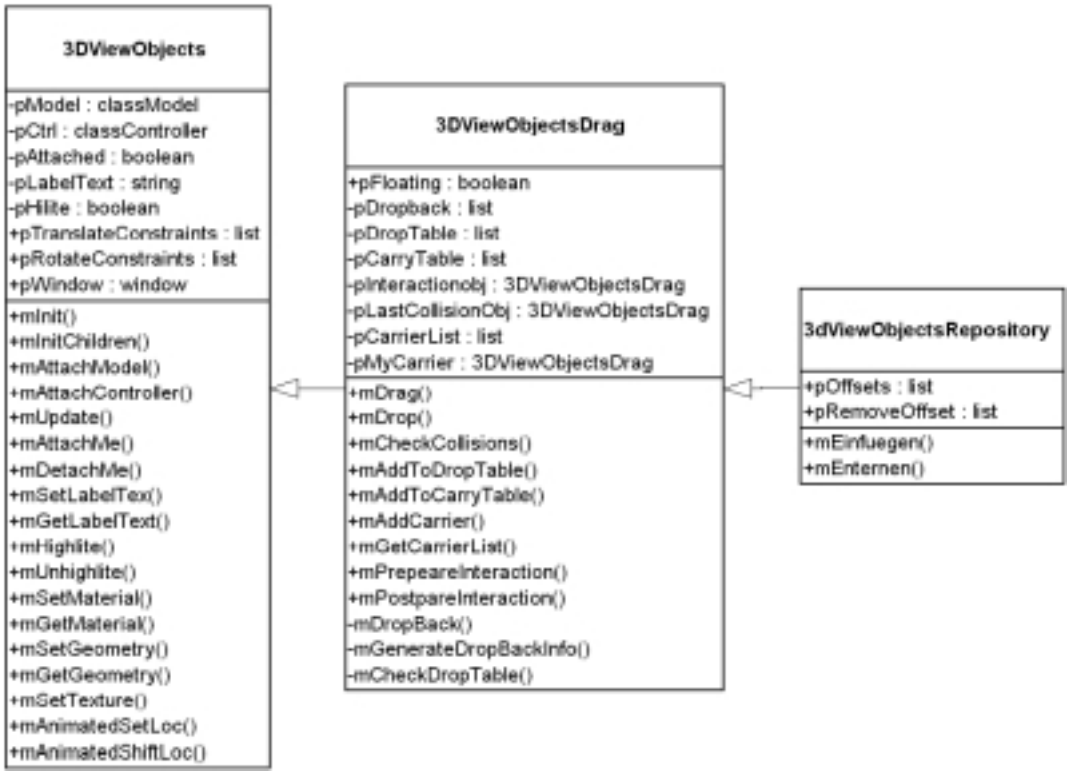

Abbildung 4.9: Das GENLAB-Framework wird durch diese abstrakten View3D-Klassen zur Erstellung von echten 3D-Objekten erweitert

Schnittstelle bzw. von 3D DREAMS unterstützt. Lediglich die initiale Position und Orientierung werden in Attributen der Klasse `Camera3D` gespeichert. So können bei der Erstellung eines Labors mehrere Kameras an bestimmten Positionen, beispielsweise über jeden Arbeitsplatz eine, platziert werden. Beim Wechsel der Arbeitsfläche durch den Labormanager, wird die entsprechende Kamera aktiviert.

Transformnode3D:

Transformationsknoten werden von Objekten verwendet, deren Transformationen relativ zu einer bestimmten Position oder Orientierung im Raum durchgeführt werden müssen. Außer den Methoden, die durch die Schnittstelle bereits definiert sind, gibt es keine weiteren.

World3D:

Das Weltobjekt dient nur der Organisationen von Szenen. Es hat weder eine Position noch eine Orientierung. In virtuellen Laboren können Weltobjekte dazu verwendet werden, verschiedene Räume zu erstellen. Bei der Darstellung der Szene werden nur die Objekte berücksichtigt, die sich in der Welt mit der aktiven Kamera befinden. Das Weltobjekt besitzt daher keine weitern Operationen und Attribute, ausser denen der Schnittstellenklasse.

Controller

Aufgrund der in Abschnitt 4.2.3 beschriebenen Designentscheidung sind die Controller-Klassen des GENLAB-Frameworks nicht in der 3D-Visualisierung verwendbar. Einige Methoden der Controller können jedoch auf die Methoden neuer für die 3D-Visualisierung erstellten Controller übertragen werden, denn die Aktion, die bei einer Interaktion des Nutzers auf einen View ausgelöst wird, ist oft unabhängig von der Dimension des Views dieselbe, nämlich eine Benachrichtigung an das Modell. Direkte Manipulationen des Views vom Controller müssen an die dritte Dimension angepasst werden. Die Basisklasse der 3D-Controller `3DCTRL` stellt die grundlegenden Mechanismen zur Steuerung der Nutzerinteraktion zur Verfügung:

3DCtrl:

Damit die Kontrolle des Nutzers über einen View präzise gesteuert werden kann, werden Nutzerinteraktionen wie im GENLAB-Framework in verschiedene Klassen eingeteilt. Diese Klasse `3DCtrl` stellt folgende Interaktionskategorien zu Verfügung, die selektiv ein- oder ausgeschaltet werden können:

- ☞ Operate: Eine Operation auf einem View kann ausgeführt werden (z. B. das Betätigen eines Schalters)
- ☞ Move: Der View kann bewegt werden
- ☞ Rotate: Der View kann rotiert werden
- ☞ Help: Es kann Hilfe über den View abgerufen werden
- ☞ Label: Es kann ein Label zu dem View angezeigt werden

Die Kontrollen können vollständig oder nur temporär aktiviert oder deaktiviert werden. Dazu bietet die Klasse folgende Methoden:

- ☞ `mActivateControl(Kategorie)`
  Der Nutzer hat grundsätzlich die Möglichkeit den View über die Interaktionsklasse `Kategorie` zu manipulieren.
- ☞ `mDeactivateControl(Kategorie)` analog zu `mActivateControl()`
- ☞ `mHoldControl(Kategorie)`
  Die Kontrolle wird vorübergehend gesperrt.
- ☞ `mResumeControl(Kategorie)`
  Die Sperrung einer Kontrolle wird wieder aufgehoben.
- ☞ `mGetControl(Kategorie)`
  Gibt zurück, ob der Nutzer den View mittels der Interaktion `Kategorie` zur Zeit kontrollierin kann.

Der Controller eines Views empfängt alle Nutzerevents auf diesem View – vorausgesetzt der View ist nicht angehängt (attached), unabhängig davon ob der konkrete Controller einen Event verwerten kann oder nicht. Darum werden in dieser Basisklasse alle möglichen Nutzerevents außer `mMouseDown()` abstrakt definiert. (In diesem Visualisierungskonzept nur Mouseevents):

- `mMouseUp`
- `mRightMouseDown()`
- `mRightMouseUp()`
- `mMouseLeave()`
- `mMouseEnter()`
- `mMouseWithin()`
- `mMouseUpOutside()`

Die einzige in dieser Klasse implementierte Mouseevent-Methode ist die Methode `mMouseDown()`. Sie ermöglicht eine direkte Rotation oder Translation eines 3D-Objektes durch den Anwender und wird als Basisoperation für jeden Controller zur Verfügung gestellt.

Interaktive Transformation eines 3D-Objektes

Eine interaktive Transformation kann nur unter bestimmten Vorkehrungen durchgeführt werden:

- **Wahl des Transformationsmodus**
  Allein die Mausinformation reicht für die Durchführung der direkten Manipulation nicht aus, denn es kann noch nicht erkannt werden, ob der Nutzer das Objekt drehen oder verschieben will. Die häufigste Transformationsmethode in virtuellen Laboren ist die Translation. Standardmäßig befindet sich der Nutzer deshalb im Drag-Modus, d. h. die Mausbewegungen werden in Bewegungen des Views umgesetzt. Um ein Objekt zu rotieren, muss der Nutzer in den Rotationsmodus wechseln. Dazu gibt es mehrere Möglichkeiten. In 3D-Grafikprogrammen und Quicktime-Filmen kann dieser Moduswechsel durch Betätigung von grafischen Schaltern vollzogen werden. Diese Vorgehensweise eignet sich insbesondere in 3D-Viewern, bei denen Moduswechsel selten notwendig sind. Das Erreichen dieser Buttons erfordert jedoch weitere Mausbewegungen und verlangsamt den Interaktionsprozess. Gerade in virtuellen Laboren kann häufig ein Wechsel des Modus notwendig sein. Aus diesem Grunde wird in der Controllerbasisklasse zum Wechsel des Modus standardmäßig eine Zusatztaste definiert, die in den Rotationsmodus schaltet. Nach dem Loslassen dieser Zusatztaste wird in den Drag-Modus zurückgesprungen. Der Entwickler behält jedoch die Freiheit die Tasten zum Moduswechsel zu redefinieren. Ebenso kann die Routine, die den aktuell gesetzten Modus abfragt, überschrieben werden, so dass ein Moduswechsel via grafischer Buttons ebenfalls leicht realisiert werden kann. Weiterhin können diese Wechsel zwischen verschiedenen Controllern unterschiedlich definiert werden. Views besitzen unterschiedliche Operationsmöglichkeiten. Es ist z. B. nicht sinnvoll, eine Tür erst durch die Betätigung einer Zusatztaste drehen zu können, denn die einzige gültige Operation für die Tür ist die Drehung (der Anwender wird nicht denken, er könne die Tür bewegen). Aus diesem Grund kann der Entwickler in jeder abgeleiteten Controllerklasse eigene Tastenkombinationen

und Standardwerte für den Wechsel zwischen Rotations- und Bewegungsmodus definieren. So wird ermöglicht, dass beispielsweise eine Tür der Mikrowelle ohne Betätigen einer Zusatztaste gedreht werden kann.

- ☞ **Bestimmung der Rotationsachsen und Bewegungsrichtungen**
  Neben den Modi Bewegung und Rotation, müssen zwei weitere Modi existieren, die bestimmen in welcher Richtung die Bewegung bzw. um welche Achsen die Rotation stattfinden soll. Eine Translation kann in drei Richtungen durchgeführt werden (in X, Y und Z). Mit einer 2D-Maus alleine kann der Anwender nur zwei Richtungen vorgeben. Um ein Objekt in die dritte Richtung zu verschieben, muss in einen anderen Modus geschaltet werden. Analoges gilt für die Rotation, die um drei Achsen getätigt werden kann. Die Wahl dieser Modi kann analog zum Wechsel des Rotations-Bewegungs-Modus umgesetzt werden.
- ☞ **Einschränkung der Rotation und Bewegung**
  Nicht immer sollen 3D-Objekte vollkommen frei bewegt oder rotiert werden können. Zum einen könnte sich der Benutzer durch die vielen Interaktionsmöglichkeiten „verirren“, d. h. ihm ist es nicht mehr möglich, beispielsweise ein Gerät so zu rotieren, wie er es gerne möchte. Zum anderen gibt es Laborkomponenten, die nur in bestimmte Richtungen gedreht werden sollen. Ein mit einer Flüssigkeit gefüllter Messbecher sollte in manchen virtuellen Laboren sicherlich nicht auf den „Kopf“ gestellt werden können, da dann entweder der Fall der Flüssigkeit korrekt modelliert und dargestellt werden müsste oder ein unrealistischer Eindruck entsteht. Der Controller braucht daher Kenntnisse über diese Transformationseinschränkungen. Jeder View bestimmt für sich selbst inwieweit er bewegt oder rotiert werden kann. Diese Informationen übergibt der View seinem Controller bei der Erzeugung als Parameter. Auf diese Weise kann eine Controllerklasse mehrfach genutzt werden, auch wenn die Viewobjekte verschiedene Einschränkungen besitzen. Bei einer Transformation müssen die angegebenen Grenzen überprüft und die Transformationswerte gegebenenfalls auf den entsprechenden Randwert gesetzt werden.

Von der Klasse `3DCtrl` können einfache Klassen abgeleitet werden, die lediglich einige Kontrollen festlegen. So existieren in Laboren oft Objekte, die entweder nur bewegt, nur rotiert oder rotiert und bewegt werden können. Für diese Zwecke werden die Klassen `3DCtrlMove`, `3DCtrlRotate` und `3DCtrlMoveRotate` erstellt.

Die bisher beschriebenen Controller-Klassen reichen aus, um allgemeingültige direkte Manipulationen (Bewegung und Rotation) eines 3D-Objektes durchzuführen. Für spezielle Operationen, die Modelländerungen nachsichziehen, werden weitere Klassen benötigt, die an ihre jeweiligen Bedürfnisse angepasste Methoden beinhalten.

Module

Die oberste Klasse der Module aus dem GENLAB-Framework (`classModule`) kann bei der Verwendung der 3D-Visualisierung nur mit kleinen Änderungen verwendet

werden. Deshalb wird direkt von der Klasse `classModule` eine neue Klasse `classModule3D` abgeleitet, welche die für die 3D-Visualisierung wichtigen Operationen vornimmt. Die `mDisplay()`-Routine von `classModule` muss überschrieben werden, da die Reihenfolge der Methodenaufrufe durch die Integration des 3D DREAMS Viewport nicht mehr gültig ist. Das hängt damit zusammen, dass der Viewport erst erzeugt *und* dargestellt werden muss, bevor überhaupt neue Laborkomponenten, also 3D-Objekte erzeugt werden können. Die Display-Routine von `classModule` kann in die neue Klasse übernommen werden. Am Ende dieser Routine muss jedoch dafür Sorge getragen werden, dass die 3D-Laborkomponenten erzeugt, initialisiert und dargestellt werden. Dies übernimmt die neue Methode `mDisplay3d()` der Klasse `classModule3D`. Da der Viewport als normales 2D-Viewobjekt integriert wurde, ist dieser nach Ablauf der „herkömmlichen“ `mDisplay()`-Routine bereits dargestellt. Die Methode `mDisplay3D()` ruft nacheinander die folgenden Methoden auf:

① `mCreateLaborComponents3D()`

② `mSetLaborComponentsLocation3D()`

③ `mInitLaborComponents3D()`

④ `mInitialPrepareActions3D()`

Sie sind analog zu den entsprechenden Routinen des GENLAB-Frameworks zu verstehen.

Die direkten Unterklassen von `classModule` müssen aufgrund der neuen Klasse `classModule3D` und dadurch verursachten Hierarchieänderung ebenfalls neu angepasst werden. Ihre Implementation braucht nicht verändert werden, da die in ihnen erzeugten GUI-Komponenten übernommen werden können. Lediglich ihr Vorfahre wird geändert. Damit sowohl 2D-Versuche als auch 3D-Versuche mit dem Framework möglich sind, wird die bestehende Hierarchie nicht geändert, sondern für die zu ändernden Klassen (`classModuleVersuch`, `classModuleStandalone` und `classSkill`) äquivalente Klassen erstellt. Für Module mit 3D-Integration ergibt sich damit die in Abbildung 4.10 dargestellt Klassenhierarchie.

Die konkreten Module zur Erzeugung und Initialisierung der benötigten Laborkomponenten müssen neu implementiert werden. In ihnen werden die konkreten Methoden `mCreateLaborComponents3D()`, `mSetLaborComponentsLocation3D()` und `mInitialPrepareActions3D()` implementiert, die von der Methode `mDisplay3D()` der Klasse `classModule3D` aufgerufen werden.

Labormanager

Die Methoden des Labormanagers können weitestgehend verwendet werden. Anpassungen werden in einer abgeleiteten Klasse `labormanager3D` vorgenommen. Geändert werden muss die Methode `mSetWorkspace()`. Sie wird daher in der Klasse `labormanager3D` redefiniert. Zum Wechseln eines Workspaces wird lediglich die entsprechende Kamera aktiviert.

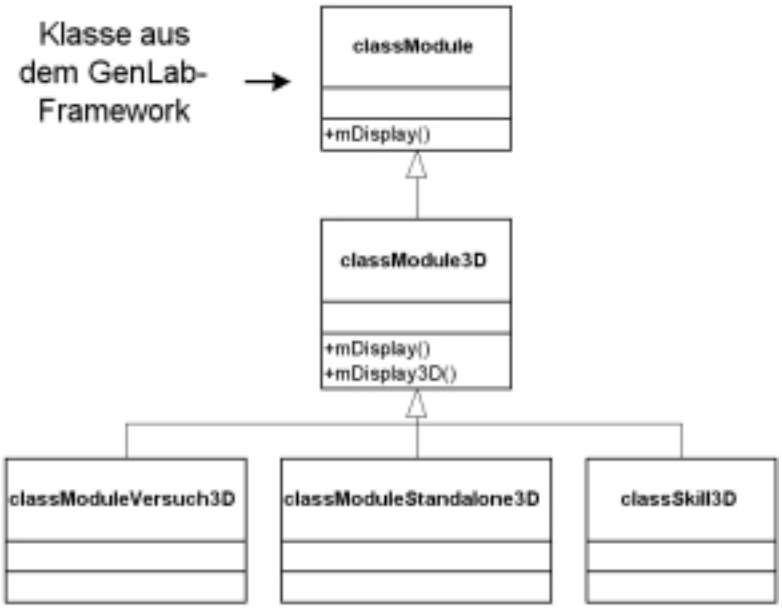

Abbildung 4.10: Erweiterung des GENLAB-Frameworks um abstrakte Klassen zur Erstellung neuer Module mit 3D-Untersützung

GUI-Komponenten

Die GUI-Komponenten können größtenteils weiterverwendet werden, da sie als 2D-View über den Viewport gelegt werden können. Die meisten GUI-Komponenten können weiterhin auf 2D-Basis existieren. Die Objektbar, die im GENLAB-Framework z. T. als GUI-Komponente behandelt wird, muss als 3D-View-Objekt implementiert werden, wenn 3D-Objekte auf die Objektbar per Drag & Drop gezogen werden sollen. 3D-Objekte können nicht mit 2D-Objekte interagieren, da 3D-Objekte in diesem Visualisierungskonzept nicht durch Sprites repräsentiert werden und nicht mit 2D-Spirtes kollidieren können.

Ablaufsteuerung

Die Versuchssteuerung des GENLAB-Frameworks überwacht die Zustände der Laborkomponenten und reagiert entsprechend auf nicht erlaubte Aktivitäten, erfolgreich durchgeführte Arbeitsschritte etc. Dazu brauchen nur die Modelle der Laborkomponenten beobachtet werden. Zum Vorführen bestimmter Arbeitsschritte kann die Versuchssteuerung auch direkt auf View-Objekte zugreifen, um beispielsweise Positionsänderungen eines Objektes durchzuführen. Wurde eine konkrete Versuchssteuerung für ein Modul auf 2D-Basis implementiert, können die Objekte der Versuchssteuerung mit direktem Zugriff auf den 2D-View in einer 3D-Visualisierung nicht verwendet werden. Die Manipulationen des Views durch die Versuchssteuerung muss an die 3D-Visualisierung angepasst werden.

4.2.4 Bausteinkomponenten

Einzelne Bestandteile von Laborkomponenten sind oft in verschiedener Weise auf oder in anderen Laborkomponenten wiederzufinden. In physikalischen Laboren exisitieren beispielsweise Schalter aller Art häufig auf unterschiedlichen Geräten. Dazu zählen Tastschalter, Kippschalter, kontinuierliche und stufenweise Drehschalter. In Chemielaboren wiederum existieren Skalen verschiedener Ausprägungen auf unterschiedlichen Behältern. Die meisten modernen elektrischen Geräte besitzen digitale Anzeigen. Wie [Elf99] beschreibt können solche häufig vorkommenden Elemente als *Bausteine* in das Framework integriert werden, um den Entwicklungsprozess für virtuelle Labore zu vereinfachen. Solche Bausteine können aus vorgefertigten, aber flexiblen und parametrisierbaren Models, Views und Controllern bestehen. Ihre Views können einfach in einen „Vaterview“ integriert werden.

Damit die Entwicklung virtueller dreidimensionaler Labore ebenfalls erleichtert wird, kann das Baustein-Prinzip problemlos in ein 3D-Visualisierungskonzept übertragen werden. Der Vorteil gegenüber den Bausteinen in 2D ist, dass z. B. für verschiedene Schalterstellungen in der 3D-Visualisierung lediglich eine Transformation durchgeführt werden muss. Im 2D-Labor können diese nur durch Austauschen entspechender Bitmaps, die vorher gefertigt werden müssen, simuliert werden. In Kapitel 5 werden exemplarisch verschiedene Bausteine implementiert.

4.3 Zusammenfassung

Der Entwurf der 3D-Visualisierung wurde in zwei Teile unterteilt. Im ersten Teil fand eine Analyse der zu entwerfenden Aufgabe auf sehr abstrakter Ebene statt. Diese Analyse gliederte die für die 3D-Visualisierung wichtigen Aufgaben in Komponenten, die möglichst unabhängig voneinander agieren und ausgetauscht werden können. Als Komponenten kristalisierten sich die *Render-Engine* zur Berechnung, Darstellung und Verwaltung der 3D-Objekte, die *Kollisionserkennung* zur Ermittlung von Kollisionen und das eigentliche virtuelle Labor (*Kernlabor*) heraus. Weiterhin wurden im ersten Teil des Entwurfs Schnittstellen entworfen, über die jede Kommunikation zwischen den Komponenten stattfindet. Auf diesem Wege wird eine größtmögliche Kapselung gewährleistet.

Der zweite Teil des Entwurfs geht detailliert auf die Anbindung der 3D-Visualisierung an das GENLAB-Framework unter Verwendung der Entwicklungstools DIRECTOR und 3D DREAMS ein. Der Viewport von 3D DREAMS wurde als 2D-View-Objekt unter Verwendung der GENLAB-Framework -Klassen in die DIRECTOR-Bühne integriert, wodurch die Grundlage für die Darstellung dreidimensionaler Szenen geschaffen wurde. Auf diese Weise können die zweidimensionalen GUI-Elemente aus dem GENLAB-Framework weiterverwendet werden. Es wurde ein Mechanismus erstellt, der die Weiterleitung der Events auf dem Viewport an die entsprechenden 3D-Viewobjekte weiterleitet. Jedes 3D-Objekt kann dadurch die gleichen Nutzerereignisse wie die 2D-Objekte aus GENLAB empfangen und verarbeiten. In Abschnitt 4.2.3 wurden die

GENLAB-Framework -Klassen näher untersucht und auf Verwendung geprüft. Einige Klassen, unter anderem die Modelle, können in der 3D-Visualisierung direkt genutzt werden. Andere Klassen konnten durch Redefinition einiger Methoden an das 3D-Visualisierungkonzept angepasst werden. Es wurde weiterhin festgestellt, dass weder Viewklassen noch Controllerklassen auf dem GENLAB-Framework verwendet werden können. Deshalb wurden neue abstrakte View- und Controllerklassen zur Erstellung virtueller Labore entworfen. Die abstrakten Controllerklassen werden in Abbildung 4.11 dargestellt. Einen Überblick über die neuen Viewklassen und deren Anbindung an die Schnittstellen gibt Abbildung 4.12.

Abbildung 4.11: Übersicht über die neue Controller-Struktur

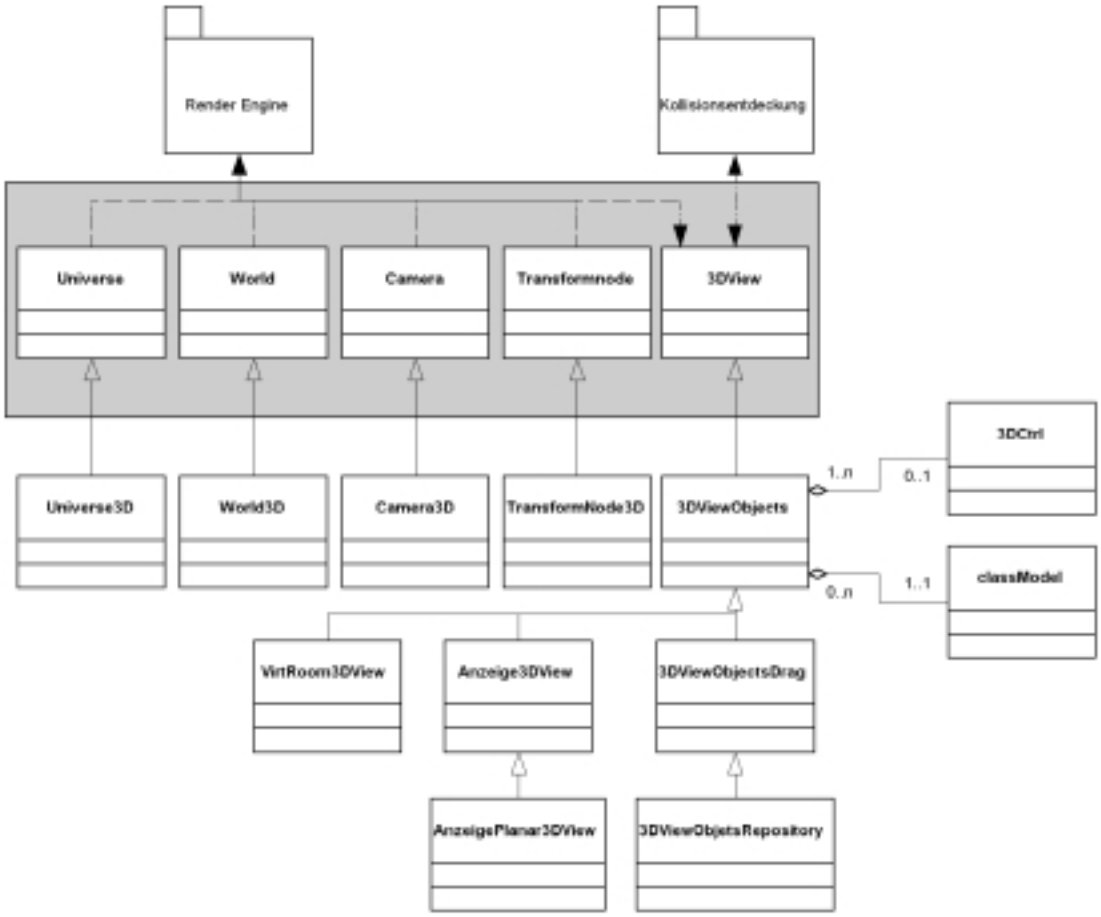

Abbildung 4.12: Übersicht über die neue View-Struktur

5 Implementierungsaspekte

In diesem Kapitel werden die technischen Details der Implementierung des in Kapitel 4 vorgestellten Entwurfs beschrieben. Nach Vorstellung der verwendeten Hard- und Software werden in Abschnitt 5.2 der Import und die dynamische Erzeugung interaktiver 3D-Objekte erläutert. Anschließend wird die Weiterleitung der Anwendereingaben an entsprechende 3D-Objekte verdeutlicht. Es folgen Details zur interaktiven Transformation eines 3D-Objektes durch den Anwender sowie zur Behandlung von Interaktionen zwischen Views. Wiederverwendbare, parametrisierbare Bausteinkomponenten werden in Abschnitt 5.6 erörtert. In den letzten beiden Abschnitten werden Maßnahmen zur Steigerung der Performance und 3D DREAMS - spezifische Verbesserungen vorgestellt.

5.1 Implementierungsumgebung

Die Implementierung des im Kapitel 4 vorgestellten Entwurfs erfolgte unter folgender Umgebung:

- ☞ MACROMEDEDIA DIRECTOR SHOCKWAVE STUDIO, Version 8.0
- ☞ 3D DREAMS FOR DIRECTOR VON SHELLS INTERACTIVE, Release Candidate 2.5
- ☞ GENLAB-FRAMEWORK vom 01.12.2000
- ☞ Weitere Xtras für DIRECTOR, die jedoch ausschließlich vom GENLAB-Framework genutzt wurden
- ☞ MICROSOFT WINDOWS 98
- ☞ DIRECTX 7
- ☞ INTEL PENTIUM III, 450 Mhz, 192 MB Arbeitsspeicher
- ☞ Grafikkarte: 3DFX VOODOO 3 3000, 16 MB
- ☞ CINEMA 4D XL R6 von MAXON zur Erstellung der 3D-Modelle und zum Export in das 3D-STUDIO-Format (3ds)

Die Programmierung fand ausschließlich in DIRECTORS Skriptsprache LINGO statt. Diese Sprache unterstützt die grundlegenden objektorientierten Konzepte, jedoch keine Mehrfachvererbung.

5.2 Erzeugung manipulierbarer und interaktiver dreidimensionaler Objekte

Die Integration interaktiver dreidimensionaler Objekte in virutelle Labore erfordert einen Zugriff auf die Daten der 3D-Modelle (Geometrie, Material und Erscheinung). Ausserdem müssen sie mit interaktiven kontextabhängigen Fähigkeiten ausgestattet, also mit Models, Views und Controllern „verbunden“ und dynamisch erzeugt werden können.

Zugriff von 3D DREAMS auf das 3D-Modell

Alle Texturen werden in DIRECTOR importiert und in Castmember abgespeichert. 3D DREAMS kann über diese Castmember auf die Texturdaten zugreifen. Der Zugang zu den Geometrie- und Materialdaten kann in 3D DREAMS über zwei verschiedene Möglichkeiten erfolgen:

① direkter Zugriff auf die Szenendatei im 3D STUDIO-Format. Die Referenzierung erfolgt über Verknüpfungen.

② Import der notwendigen Daten und Abspeicherung in DIRECTOR-Castmember. Der Zugriff auf das 3D-Modell erfolgt nicht mehr über die Datei, sondern über die Castmember

Die erste Möglichkeit bietet den Vorteil, dass beim Authoring stets auf aktuellen Daten gearbeitet wird, auch wenn sich das 3D-Modell ändert. Spätestens bis zur Distribution muss das 3D-Modell komplett in DIRECTOR importiert werden, da 3D DREAMS aus einem SHOCKWAVE-Film oder Projektor nicht mehr auf das Dateisystem zugreifen kann.

Zur Erstellung der Verknüpfungen stellt 3D DREAMS ein GUI zur Verfügung. Der endgültige Import der Daten erfolgt über einen mitgelieferten DIRECTOR-Film. Die importierten Daten werden binär in spezielle Castmember vom Typ `#shellsBinary` gespeichert.

Szenenverwaltung und Referenzierung der 3D-Modelle

Die Verwaltung der Hierarchie der 3D-Szene übernimmt 3D DREAMS. Sie kann über ein GUI vom Entwickler verändert werden. Zur einfachen Referenzierung eines kompletten aus mehreren Einzelteilen bestehendes 3D-Modell empfiehlt sich eine gemein-

same Wurzel. Abbildung 5.1 zeigt eine sinnvolle Hierarchie der Mikrowelle. Als Vaterelement dient der Grundkörper der Mikrowelle, alle weiteren Bestandteile werden relativ zum Grundkörper positioniert und ausgerichtet. Die Erzeugung, Initialisierung, Transformationen usw. können über die Wurzel durchgeführt werden, die entsprechende Aufgaben an ihre Kindobjekte weiterleitet.

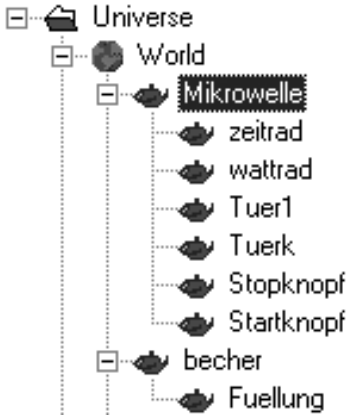

Abbildung 5.1: Hierarchie einer Szene in 3D DREAMS

Damit ein 3D-Modell von Lingo aus referenziert werden kann, muss der Entwickler dem 3D-Modell über das GUI eine globale Variable zuweisen. Wurde beispielsweise die Hierarchie der Mikrowelle wie in Abbildung 5.1 gewählt, reicht es aus, nur dem 3D-Element „Mikrowelle“ eine globale Variable zuzuweisen. Da jedes 3D-Modell Kenntnisse über seine Unterobjekte besitzt, kann die Referenzierung der Unterobjekte über das Hinabsteigen in der Szenenhierarchie erfolgen.

Austattung von Fähigkeiten zur Interaktion und Manipulation, Anbindung der Viewobjekte an 3D-Modelle

Die bisher importierten 3D-Modelle besitzen noch keine Möglichkeit, auf Anwendereingaben reagieren zu können. Zur Manipulation (z. B. Positions-, Materialänderungen) können über die globale Variable, die einem 3D-Modell zugewiesen wurde, nur direkt Methoden von 3D DREAMS verwendet werden. Ein primäres Ziel des Entwurfes ist aber die Kapselung der Methodenaufrufe durch die Schnittstelle `3DView` (siehe Abbildung 4.5). Es muss also eine Möglichkeit geschaffen werden, die Schnittstellenklasse `3DView` an die 3D-Objekte anzubinden und Nutzerevents an entsprechende Controller weiterzugeben. Alle 3D-Objekte in 3D DREAMS sind Instanzen von der 3D DREAMS - Klasse `i3DDObj`. Es besteht daher die Möglichkeit durch Vererbung, diese Klasse zu verfeinern und eigene Methoden zu implementieren. So können kontextabhängige Manipulationen durchgeführt und Nutzerevents an einen Controller weitergeleitet werden. Als direkter Nachfahre der Klasse `i3DDObj` kann die Schnittstellenklasse `3DView` abgeleitet werden. Dadurch können die Laborkern-Viewklassen an die 3D-Modelle „angeknüpft“ werden. Für die Mikrowelle beispielsweise ergibt sich die in Abbildung 5.2 gezeigte Viewhierarchie.

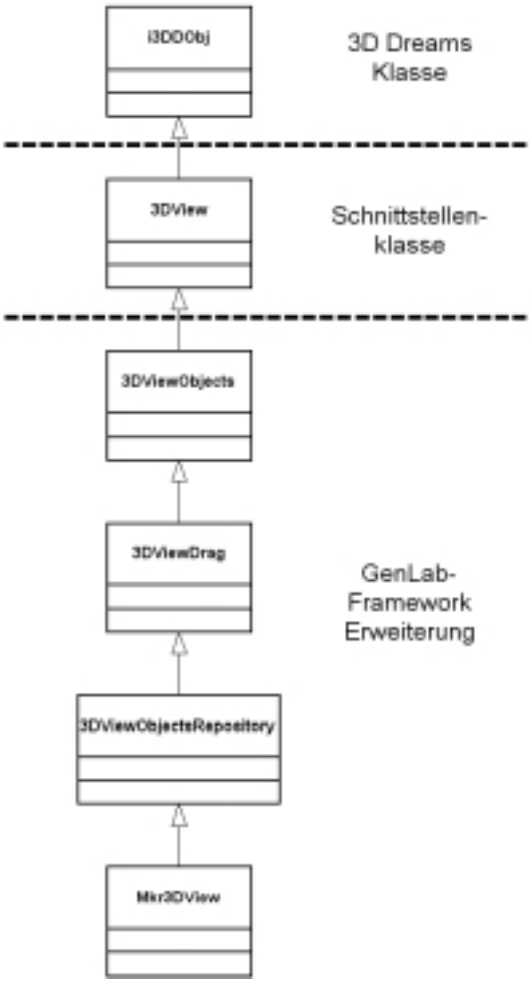

Abbildung 5.2: Viewhierarchie der Mikrowelle

Durch den Aufbau dieser Hierarchie ist die Klasse `mkr3DView` noch an kein spezielles 3D-Modell gebunden. Sie kann praktisch für jedes 3D-Modell verwendet werden. Diese Bindung kann nur durch das GUI von 3D DREAMS erfolgen, indem dem spezifischen 3D-Modell das entsprechende Skript zugewiesen wird (siehe Abbildung 5.3). Der so erzeugte Mikrowellenview wird in der globalen Variablen `mkrViewMaster` gespeichert.

Dynamische Erzeugung

Die Darstellung der 3D-Modelle und die Erzeugung deren zugewiesenen Skripte wird sofort bei der Initialisierung des Viewports, also sobald der Viewport auf der DIRECTOR-Bühne erscheint, von 3D DREAMS durchgeführt. Der Entwickler wird also dadurch eingeschränkt, dass er den Zeitpunkt der Erzeugung eines Objektes nicht selbst bestimmen kann. Weiterhin ist es auf diese Weise nicht möglich, eine zur Entwicklungszeit unbekannte Anzahl von Instanzen eines 3D-Objektes anzulegen, um beispielsweise mehrere gleiche Messbecher oder Pipetten dem Labor hinzuzufügen. Zur

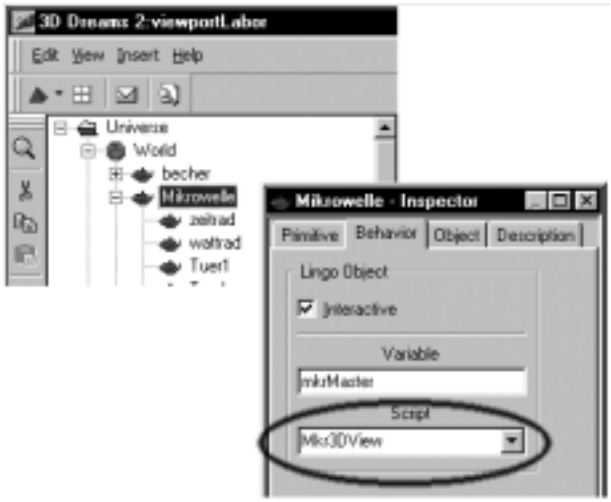

Abbildung 5.3: Zuweisung eines View-Skriptes an ein 3D-Modell

Lösung des Problems bietet sich folgende Vorgehensweise zur dynamischen Erzeugung an:

① Jedes benötigte 3D-Modell wird genau einmal zur 3D-Szene hinzugefügt und mit dem entsprechenden View-Skript versehen. Diese Objekte werden im folgenden *Masterobjekte* genannt.

② Die Masterobjekte werden so positioniert, dass sie durch den Anwender niemals gesehen werden können.

③ Wird ein bestimmtes 3D-Objekt im Labor benötigt, wird von dem entsprechenden Masterobjekt eine Kopie erzeugt, ausgerichtet und positioniert. Für das Erstellen einer Kopie stellt 3D DREAMS eine Funktion zur Verfügung.

Auf diese Weise kann der Entwickler dynamisch neue 3D-Objekte erzeugen und in die Laborszene integrieren. Die Erzeugung erfolg durch folgenden Aufruf:

```
<viewobjekt> = <masterObjekt>.mNew()
```

Eine Mikrowelle, deren Masterobjekt durch die globale Variable `mkrMaster` referenziert wird, kann durch folgenden konkreten Aufruf dynamisch erzeugt werden:

```
mkrView = mkrMaster.mNew()
```

Die Methode `mNew()` befindet sich in der Schnittstellenklasse `3DView` und sieht folgendermaßen aus:

```
on mNew me
  tView = me.deepClone3d()
  return tView
end
```

Die Methode `deepClone3d()` stammt aus einer 3D DREAMS - Klasse. Sie sorgt für das Duplizieren des 3D-Objektes samt seiner Kindobjekte. Die Referenz auf das neue 3D-Objekt wird zurückgeliefert.

Die Methode `deepClone3d()` ermöglicht es nicht, den neu erzeugten Viewklassen, Parameter zu übergeben. Eine Parametrisierung eines Views kann nur durch Aufruf einer weiteren durch den View zur Verfügung gestellten Methode direkt nach der dynamischen Erzeugung geschehen. Die Methode heisst `mInit()`.

5.3 Weiterleitung der Anwendereingaben an 3D-Objekte

Im zweiten Teil des Entwurfes (Abschnitt 4.2.2 auf Seite 64) wurde bereits der Mechanismus zur Generierung und Aussendung der Nutzerevents an die entprechenden Viewobjekte abstrakt entworfen. Der Mechanismus wird in dem Controller für den Viewport implementiert. Da der Viewportcontroller unter Verwendung der GENLAB-Frameworkklassen erstellt wird, empfängt er alle Events, die jeder andere „2D-Controller“ erhalten würde. Auf Basis dieser Events können die konkreten auf ein 3D-Objekt ausgeübten Nutzerinteraktionen hergeleitet und an die beteiligten 3D-Viewobjekte gesendet werden. Das folgende Listing zeigt die Implementierung des Viewportcontrollers (`classControllerUniverse`):

```
01 property ancestor, pActObject, pMouseDownObject

02 on new me, dModel, dMember
03   ancestor = new(script "classControllerBitmap", dModel, dMember, 36)
04   ancestor.mDropIgnore()
05   me.pActObject = VOID
06   me.pMouseDownObject = VOID
07   me.pbgColor = rgb(255,255,255)
08   return me
09 end

10 on mMouseWithin me

  -- events mMouseEnter, mMouseLeave, mMouseWithin
  -- an die richtigen 3D-Objekte senden

11   tNewObject = me.pSuperView.mObjectAt(the mouseLoc[1],\
                                          the mouseLoc[2])

12   if tNewObject <> me.pActObject then
13     if objectP(me.pActObject) then
14       me.pActObject.mMouseLeave()
15     end if
```

```
16     me.pActObject = tNewObject
17     if objectP(me.pActObject) then
18       me.pActObject.mMouseEnter()
19       me.pActObject.mMouseWithin()
20     end if
21   else
22     if objectP(me.pActObject) then
23       me.pActObject.mMouseWithin()
24     end if
25   end if
26 end

27 on mRightMouseDown me
    -- Datenbankaufruf
28 end

29 on mRightMouseUp me
    -- Datenbankaufruf
30 end

31 on mMouseLeave me
32   if objectP(me.pActObject) then
33     me.pActObject.mMouseLeave()
34     me.pActObject = VOID
35  end if
36 end

37 on mMouseEnter me
38  nothing
39 end

40 on mMouseDown me
41   if objectP(me.pActObject) then
42     me.pMouseDownObject = me.pActObject
43     me.pActObject.mMouseDown()
44   end if
45 end

46 on mMouseUp me
47   if objectP(me.pMouseDownObject) then
48     if me.pMouseDownObject<>me.pActObject then
49       me.pMouseDownObject.mMouseUpOutside()
50     end if
51     me.pMouseDownObject = VOID
52   end if
53   if objectP(me.pActObject) then
54     me.pActObject.mMouseUp()
55   end if
56 end
```

```
57 on mMouseUpOutside me
58   if objectP(me.pMouseDownObject) then
59     me.pMouseDownObject.mMouseUpOutside()
60     me.pMouseDownObject = VOID
61   end if
62 end
```

Die Events `mMouseEnter, mMouseLeave` und `mMouseWithin` für die 3D-Objekte werden in der Methode `mMouseWithin()` des Viewports generiert (Zeilen 10-26). Diese Methode wird solange wiederholt durchlaufen, wie sich der Mauszeiger innerhalb des Viewports befindet. Über die Methode `mObjectAt()` wird das jeweils unter dem Mauszeiger befindliche 3D-Objekt ermittelt (Zeile 11) und mit dem aus dem letzten Durchgang aktuellen Objekt verglichen (Zeile 12). Unterscheiden sich die Objekte und ist das vorherige Objekt gültig (nicht VOID), dann wird diesem der Event `mMouseLeave` gesendet (Zeile 14), denn der Mauszeiger befand ich in dem vorigen Durchgang auf dem Objekt, mittlerweile aber nicht mehr. Das gerade unter dem Mauszeiger befindliche 3D-Objekt wird in der Property `pActObject` gespeichert. Ist dieses Objekt nicht VOID werden ihm die Events `mMouseEnter` und `mMouseWithin` zugesendet (Zeilen 22 und 23). Für den Fall, dass das gerade unter dem Mauszeiger befindliche Objekt das gleiche Objekt wie im vorigen Durchgang ist, wird der Event `mMouseWithin` an dieses 3D-Objekt gesendet (Zeile 23). Verlässt der Mauszeiger den Viewport, erhält das aktuelle 3D-Objekt einen `mMouseLeave`-Event (Zeilen 31-36). Wird eine Maustaste gedrückt, wird dem aktuell gewählten Objekt ein `mMouseDown`-Event zugesendet (Zeile 43). Außerdem wird es in der Property `pMouseDownObject` gespeichert (Zeile 42), damit es, auch wenn der Mauszeiger das Objekt verlässt, referenziert und evtl. ein `mMouseUpOutside`-Event weitergeleitet werden kann. Bei einem Loslassen der Maustaste auf dem Viewport wird geprüft, ob das gerade aktuelle 3D-Objekt nicht dem 3D-Objekt entspricht, auf dem der Mausklick ausgeübt wurde (Zeile 48). In diesem Fall wird dem in der Variablen `pMouseDownObject` gespeicherten Objekt der Event `mMouseUpOutside` gesendet (Zeile 49). Das aktuelle Objekt bekommt in jedem Fall ein `mMouseUp`-Event. In der Methode `mMouseUpOutside()` (Zeile 57-62) des Viewportcontrollers wird, wenn vorher auf ein 3D-Objekt geklickt wurde, beim Loslassen des Maustasters außerhalb des Viewports der Event `mMouseUpOutside` an dieses 3D-Objekt geliefert.

Durch diese Implementierung erhalten alle 3D-Objekte die gleichen Mouseevents, wie die 2D-Objekte, die mithilfe des GENLAB-Frameworks erzeugt wurden.

5.4 Interaktive Transformation eines 3D-Objektes

Im Entwurf wurden die notwendigen Mechanismen für eine interaktive Transformation eines 3D-Objektes analsiert (Abschnitt 4.2.3 auf Seite 70). Der Ablauf einer interaktiven Transformation wird durch das Flussdiagramm in Abbildung 5.4 beschrieben. Er

wird durchlaufen, sobald der Anwender auf ein 3D-Objekt klickt und diese Methode nicht überschrieben wurde. Sie ist im Basiscontroller `3DCTRL` implementiert.

Dieser Ablauf ermöglicht einen Wechsel zwischen Rotations- oder Bewegungsmodus während eines Drags, denn die Abfrage nach dem Modus wird erst innerhalb der Schleife durchgeführt. Die Positions- und Orientierungsaktualisierung beinhaltet eine Abfrage nach der Richtung der Bewegung bzw. der Rotationsachse und die Berücksichtigung eventueller Transformationseinschränkungen. Der View wird in diesen Routinen direkt manipuliert, d. h. die Nachrichtenkette verläuft nicht über das Modell des 3D-Objektes, denn dieses speichert keine Positionen und Ausrichtungen des Views. Folgende Aufgaben sind durch den Controller während eines Drags durchzuführen, bevor die direkte Manipulation des 3D-Objektes vollzogen werden kann:

- ☞ der aktuelle Rotations- und Bewegungsmodus muss abgefragt werden
- ☞ Rotationsachsen und Bewegungsrichtungen müssen ermittelt werden
- ☞ eine neue Position muss berechnet und mit etwaigen Transformationseinschränkungen abgeglichen werden

Abbildung 5.4: Ablauf der Methode `mMouseDown` in der Controller-Basisklasse `3DCTRL`

Abfrage und Definition des Transformationsmodus und der -richtung

Eine eindeutige Transformationsänderung eines 3D-Objektes mit der Maus kann nur durchgeführt werden, wenn sowohl der gewünschte Transformationsmodus als auch die Richtung bekannt sind. Die Methode `mTransformInfo()` führt diese Abfragen durch. Sie kann durch den Entwickler in abgeleiteten Klassen überschrieben werden. Die Methode liefert die Ergebnisse in einer einfachen Liste zurück, die aus zwei Elementen besteht: die Information über den Transformationsmodus (#drag oder #rotate) und der -Richtung in Form eines Layers (#XY, #YZ, #XZ) in der die Transformation stattfinden soll. Die Transformationsrichtung besteht deshalb immer aus zwei Achsen, da die 2D-Maus jeweils Informationen über zwei Dimensionen liefern kann. Ein Bewegung der Maus nach links/rechts oder oben/unten kann also beispielsweise in eine Bewegungsänderung des 3D-Objektes in der XY-Ebene, YZ-Ebene oder XZ-Ebene umgesetzt werden.

Zur Definition der Modiwechsel werden zwei Properties eingeführt: `pDragOption` und `pRotateOption`. Beide Variablen sind Eigenschaftlisten der Form

```
[<taste1>:<layer1>, <taste2>:<layer2>, ...]
```

`<taste1>` definiert jeweils die Zusatztaste, die gedrückt werden muss, um in den entsprechenden Modus zu wechseln. Hier kann jede Taste angegeben werden, deren Status durch Lingo überprüft werden kann. Für Sondertasten können folgende Symbole verwendet werden: `#ctrl`, `#shift` und `#option`. Außerdem kann das Symbol `#none` angegeben werden, wenn keine Taste gedrückt werden muss. `<layer1>` beschreibt die Ebene in welche die Mausbewegungen umgesetzt werden, wenn nur `<taste1>` gedrückt wird. Wird *zusätzlich* zur `<taste1>` eine weitere Taste, beispielsweise `<taste2>` betätigt, wird die Mausbewegung auf die Ebene `<layer2>` umgesetzt. Folgende Standardbelegungen werden in der Klasse `3DCTRL` vorgegeben:

```
pDragOption   = [#none:#XZ, #shift:#XY]
pRotateOption = [#ctrl:#XZ, #shift:#XY]
```

Wenn keine Taste gedrückt wird, ist der Bewegungsmodus aktiviert. Die Bewegung wird in der XZ-Ebene durchgeführt. Wenn die Shifttaste alleine betätigt wird, wird das 3D-Objekt in X- und Y-Richtung verschoben. Durch Drücken der CTRL-Taste wird in den Rotationsmodus gewechselt. Für Rotationen werden die Achsen in das Rotationskoordinatensystem (Yaw, Pitch und Roll) umgewandelt. Das Objekt wird dann also in Yaw- und Pitch-Winkeln gedreht werden. Erst durch Drücken der CTRL-Taste und der Shifttaste kann in Yaw- und Roll-Winkel rotiert werden. Ein Controller für eine Tür kann diese Belegungen überschreiben:

```
pDragOption   = [#ctrl:#XZ, #shift:#XY]
pRotateOption = [#none:#XZ, #shift:#XY]
```

So kann die Tür um Yaw und Pitch (übertragen ins Kartesische Koordinatensystem um die Y- und Z-Achse) gedreht werden, ohne das eine Taste gedrückt werden muss. Der Dragmodus muss ebenfalls neubelegt werden, damit es zu keinen Doppelbelegeungen kommt. Bei Doppelbelegungen erhält der Drag-Modus Priorität. Die Methode `mTransforminfo()` liefert anhand dieser Variablenbelegungen und der gedrückten Tasten den Transformationsmodus und die -Richtung zurück. Auf Basis dieser Informationen kann die neue Position oder Orientierung berechnet werden.

Einbeziehung und Definition der Bewegungs- und Rotationseinschränkungen

Der Entwurf der interaktiven Transformation zeigte, dass einige 3D-Objekte bestimmten Transformationseinschränkungen unterlegen sein können. Diese dienen zum einen als Hilfe für den Anwender, zum anderen um bestimmte Aktionen zu unterbinden. Die Einschränkungen werden im View als Bereiche festgelegt, in denen das 3D-Objekt rotiert oder bewegt werden darf. Diese Bereiche werden dem zugehörigen Controller übergeben. Zur Speicherung der Information werden zwei neue Properties in der Klasse `3DCtrl` definiert (`pRotateConstraints` und `pTranslateConstraints`). Die Datenstruktur der Properties ist eine Eigenschaftsliste mit drei Elementen. Die Elemente geben jeweils die untere und obere Grenze für jede Achse an. Standardmäßig sind keine Transformationen eingeschränkt. Die Properties werden daher bei der Initialisierung mit leeren Listen belegt:

```
pRotateConstraints = [#yaw:[], #pitch:[], #roll:[]]
pTranslateConstraints = [#x:[], #y:[], #z:[]]
```

Um einen gefüllten Meßbecher nur in bestimmte Bereiche rotieren zu können, kann der View dem Controller folgende Werte übergeben:

```
pRotateConstraints = [#yaw:[], #pitch:[315,45], #roll:[315,45]]
```

So kann der Meßbecher zwar ohne Einschränkungen um die Y-Achse, aber jeweils nur bis zu 45 Grad um eine der anderen Achsen gedreht werden. Die Tür eine Mikrowelle kann durch folgende Werte nur bis zu 85 Grad um ihre Y-Achse gedreht werden. Andere Rotationen werden nicht zugelassen:

```
pRotateConstraints = [#yaw:[0,85], #pitch:[0,0], #roll:[0,0]]
```

Würde einer dieser Winkelbereiche aufgrund der Mausbewegung über- oder unterschritten werden, wird der entsprechende Wert auf den Randwert des Bereiches gesetzt. Diese Berechnungen werden in der Methode `mCheckRotateConstraints` bzw. `mCheckTranslateConstraints` durchgeführt.

5.5 Interaktionen zwischen Views

Beim interaktiven Bewegen eines 3D-Objektes durch den Anwender können Interaktionen zwischen zwei Views auftreten. Es gibt zwei verschieden Interaktionsbeziehungen, die durch die Abbildung 5.5 veranschaulicht werden. Der Messbecher in der Mikrowelle stellt eine „echte“ Interaktionsbeziehung dar. Die Mikrowelle kann die Eigenschaften des Messbechers verändern. Ein weitere Beziehung stellen der Dummy-Schalter und die Mikrowelle dar. Allerdings wird der Schalter nur getragen. Diese Interaktion wird nicht auf Modellebene übertragen, so dass keine Zustandsänderungen in den Modellen stattfinden. Die Mikrowelle trägt den Schalter. Gleiches gilt für die Beziehung zwischen Pipettenspitze und Dummy-Schalter.

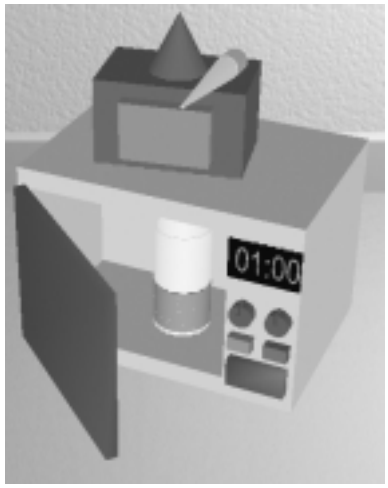

Abbildung 5.5: Interaktionsbeziehungen

Die „echte“ Interaktion wird in dieser Implementation erreicht, indem der Nutzer ein 3D-Objekt bewegt, es mit dem gewünschten Interaktionsobjekt kollidieren und wieder fallen lässt. Die reine „Trägerbeziehung“ wird erreicht, wenn der Anwender ein Objekt auf ein anderes Objekt fallenlässt, ohne dass die beiden Objekte vorher kollidierten. Jedes Objekt kann dabei selbst entscheiden, welche Beziehung mit welchem anderen Objekt erlaubt ist.

Die Aufgaben, die während einer Objektbewegung vom Programm durchzuführen sind, wurden in Abschnitt 4.2.3 auf Seite 66 entworfen und werden in der Klasse `3DViewObjectsDrag` implementiert.

Vorbereitende Maßnahmen

Bei der ersten Positionsänderung wird vom Controller die Methode `mDrag()` des Views aufgerufen. In ihr sind laut Entwurf folgende Routinen zu erledigen:

① Speicherung der notwendigen Informationen für einen evtl. Rücksprung

② Lösung bestehender Viewbeziehungen

③ Erzeugen einer Drag-Referenz

④ Aktualisierung der Hierarchie

Die notwendigen Informationen für einen Rücksprung sind das bestehende Vaterobjekt, die aktuelle Position und Orientierung. Sie werden durch die Methode `mGenerateDropBack` der Klasse `3DViewObjectsDrag` in dem Attribut `pDropBack` gespeichert. Es gibt zwei mögliche Viewbeziehungen die aufgelöst werden müssen: Befindet sich der View in einem Behälter, muss er entfernt werden. Dazu wird beim Vaterobjekt die Methode `mEntfernen()` aufgerufen. Wird das Objekt von einem anderen Objekt getragen, wir dem Trägerobjekt mitgeteilt, das diese Beziehung beendet ist. Zur Erzeugung der Drag-Referenz wird die Methode `mCreateDragReference()` aufgerufen. Das Referenzobjekt ist ein einfaches 3D-Objekt ohne Geometrie, Material und Texturdaten. Die Methode `mCreateDragReference()` ist wie folgt implementiert:

```
on mCreateDragReference me
  tDragReference = nullObjMaster.mNew()
  tOrient = me.mGetActiveCamera().mGetOrientation()
  tOrient.pitch=0.0
  tDragReference.mSetOrientation(tOrient)
  return tDragReference
end
```

Der View wird als Kindobjekt an das zurückgegebene Referenzobjekt angehängt.

Während der Bewegung

Während einer Bewegung eines 3D-Objektes durch den Anwender wird auf Kollisionen mit anderen Objekte geprüft, um mögliche Interaktionsobjekte festzuhalten. Dazu wird vom Controller die View-Methode `mCheckCollisions()` aufgerufen. Sie ist folgendermaßen implementiert:

```
on mCheckCollision me
  if me.mHasCollided() then
    tObject = me.mLastCollisionObj()
    if tObject.mPrepareInteraction(me) then
      me.pInteractionObj = tObject
    end if
  else
    me.pLastCollisionObj = VOID
    if not voidp(me.pInteractionObj) then
      me.pInteractionObj.mPostpareInteraction(me)
      me.pInteractionObj = VOID
    end if
  end if
end
```

Die Methoden `mHasCollided()` und `mLastCollisionObj()` sind Schnittstellenmethoden, welche die Kollisionserkennung starten und ein eventuelles Kollisionsobjekt zurückliefern. Wenn eine Kollision stattfand wird das passive Kollisionobjekt in der Variablen `tObject` gespeichert. Bei diesem Objekt wird die Methode `mPrepareInteraction()` mit dem gedraggten View als Parameter aufgerufen. Sie liefert zurück, ob der gedraggte View mit dem Kollisionsobjekt interagieren, beispielsweise eingefügt, kann. Ausserdem können weitere Aktionen wie ein Hilighting durchgeführt werden. Die Methode `mPrepareInteraction()` ist folgendermaßen implementiert:

```
on mPrepareInteraction me, dObj
  if me.mCheckDropTable(dObj) then
    me.mHighlite()
    me.pInteractionObj = dObj
    return TRUE
  end if
  return FALSE
end
```

Ob ein 3D-Objekt auf ein anderes 3D-Objekt gedroppt werden darf, wird in einem DropTable definiert. Das Konzept stammt aus dem GENLAB-Framework. Die Methode `mCheckDropTable(dObj)` überprüft, ob das Objekt `dObj` in der Droptabelle eingetragen ist. Wenn das der Fall ist wird die Highlite-Routine des passiven Kollisionsobjektes aufgerufen. Das Interaktionsobjekt wird in dem Attribut `pInteractionObj` bei beiden 3D-Objekten gespeichert.

Für den Fall, dass beim Bewegen des Objektes keine Kollision stattfand, die Variable `pInteractionObj` aber mit einem gültigen Objekt belegt ist, wird die Methode `mPostpareInteraction()` aufgerufen. Sie macht das Highliting wieder rückgängig. Die Variable `pInteractionobj` wird mit `VOID` belegt.

Drop

Beim Fallenlassen eines Objektes wird die Methode `mDrop()` des Views aufgerufen. Ihre Implementierung wird im folgenden dargestellt:

```
on mDrop me, dPos
  me.mDropTransform()
  if not voidP(me.pInteractionObj) then
    me.pInteractionObj.mHandleDropObject()
    me.pInteractionobj.mPostpareInteraction()
    me.pInteractionobj = VOID
  else
    me.mSetParent(me.mGetWorld())
    if not me.pFloating then
      if voidP(dPos) then
        tLoc = me.mGetLoc()
```

```
        tLoc.y=0
      end if
      me.mAnimatedSetLoc(tLoc, 10)
      if not me.pLastCollisionObj.mHandleCarrierObject(me) then
        tText = "Auf " & me.pLastCollisionObj.pViewName \
                && "können keine Objekte abgestellt werden"
        me.mFehler(tText)
        me.mDropBack()
      end if
    end if
  end if
end
```

Beim Fallenlassen des Objektes wird die Methode `mDropTransform` aufgerufen. Sie dient dem Ausrichten eines Objektes und kann dazu verwendet werden, damit beispielsweise Messbecher immer unabhängig von einer Drehung während des Drags senkrecht auf die Arbeitsfläche oder auf ein anderes Objekt abgestellt werden. Die Belegung des Attributes `pInteractionObj` ist entscheidend für den weiteren Verlauf. Bei einer gültigen Belegung können die beiden kollidierten 3D-Objekte interagieren. Beim passiven View wird dann die Methode `mHandleDropObject()` aufgerufen, in der entsprechende Interaktionaufgaben vorgenommen werden können. Das Vorgehen entspricht dem des GENLAB-Frameworks. Die Methode `mPostpareInteraction()` wird ebenfalls aufgerufen, um z. B. ein Highliting rückgängig zu machen. Für den Fall, dass es kein passendes Interaktionsobjekt gibt und das Objekt kein Floating-Objekt ist (in GENLAB z. B. die Pipette), wird das Objekt in einer Animation solange der Y-Achse entlang nach „unten“ bewegt, bis entweder der Nullpunkt erreicht oder mit einem anderen Objekt kollidiert wurde. In der Animationsroutine `mAnimatedSetLoc()` wird auf Kollisionen überprüft. Findet beim freien Fall ein Kollision statt wird analog zur Methode `mHandleDropObject()` die Methode `mHandleCarrierObject()` aufgerufen, die dem fallengelassenen Objekt mitteilt, ob es vom Kollisionsobjekt getragen werden darf. Ist das nicht der Fall, wird eine Fehlermeldung ausgegeben und ein Rücksprung auf die alte Position mit den gespeicherten Koordinaten, Ausrichtungen und der Objekthierarchie durchgeführt.

Interaktion mit Behältern Für Behälter wurde eine spezielle Klasse implementiert (`3DViewObjectsRepository`), welche die Interaktion mit gedroppten und gedraggten Objekten nach dem Entwurf (Abschnitt 4.2.3 auf Seite 68) realisiert. Die Vorgehensweise bei einem Drop entspricht der im GENLAB-Framework und wird in der Methode `mEinfuegen()` implementiert. Zuerst wird versucht das gedroppte Objekt auf Modell-Ebene in den Behälter einzufügen. Gelingt dies wird das 3D-Objekt auf View-Ebene in den Behälter integriert. Dazu werden in der konkreten Viewklasse des Behälters für jedes aufnehmbare Objekt ein Offset definiert, das die Position des gedroppten 3D-Objektes relativ zum Behälter angibt. Die Definition der Offsets beispielsweise der Mikrowelle sieht folgendermaßen aus:

```
tOffsets = [ [#x:0, #y:-4, #z:-5],[#x:-15, #y:-4, #z:-5] ]
```

Das erste gedroppte Objekt würde auf die Position (0, -4, -5), das zweite auf (-15, -4, -5) relativ zur Mikrowelle gesetzt werden. Bevor das Objekt positioniert wird ist die Kollisionserkennung für dieses Objekt auszuschalten, da ansonsten keine 3D-Objekte innerhalb der Bounding Box eines anderen 3D-Objektes gestellt werden können. Die Methode `mEinfuegen()` wurde wie folgt implementiert:

```
on mEinfuegen me, dView
  if me.pModel.mEinfuegen(dView.pModel) then
    if NOT voidP(pOffsets) then
      tOffset = pOffsets[me.pModel.pObjekte.getPos(dView.pModel)]
      dView.mSetParent(me)
      dView.mSetSolidness(FALSE)
      dView.mSetLoc(tOffset)
      dView.mSetSolidness(TRUE)
    end if
  else
    return FALSE
  end if
  return TRUE
end
```

Wie der Entwurf zeigte, weicht die Vorgehensweise beim Herausdraggen eines Objektes von der in GENLAB ab. Im konkreten View wird ein Offset zur Positionierung herausgenommener Objekte festgelegt. Der Offset ist ein einzelnes Element, da immer nur ein Objekt gleichzeitig herausgedragged werden kann. Die Methode `mEntfernen()` wurde auf folgende Weise programmiert:

```
on mEntfernen me, dView
  if not voidP(dView) then
    if me.pModel.mEntfernen(dView.pModel) then
      if not voidP(me.pRemoveOffset) then
        dView.mSetSolidness(FALSE)
        dView.mSetLoc(me.pRemoveOffset)
        dView.mSetSolidness(TRUE)
      end if
    end if
  end if
end
```

5.6 Bausteinkomponenten

In Abschnitt 4.2.4 wurden bereits die Vorteile von vorgefertigten parametrisierbaren Bausteinen erläutert. Im Rahmen dieser Arbeit wurden verschiedene Bausteinkomponenten implementiert: Schalter, dynamische digitale (Text-)Anzeigen und virtuelle nicht sichtbare Arbeitsbereiche.

5.6.1 Schalter

Ein an naturwissenschaftlichen Geräten häufig vorkommendes Element ist der Schalter. Von ihm gibt es verschiedene Ausprägungen, z. B. Taster, Drehschalter und Kippschalter. Operationen auf Schaltern bzw. Schalterbewegungen werden in GENLAB durch Austauschen von vorgefertigten Bitmaps visualisiert. Werden die Laborgeräte durch 3D-Modelle dargestellt, kann das Fertigen zusätzlicher Bitmaps für die verschiedenen Schalterstati entfallen, denn das Betätigen eines Schalters kann in der 3D-Darstellung wie in der Realität durch einfaches Bewegen und Drehen visualisiert werden.

Taster / Taste

Der in diesem Visualisierungskonzept implementierte Tasten-Baustein funktioniert folgendermaßen:

- ☞ Der Taster wird gedrückt (an eine vom konkreten View vorgegebene Position verschoben), wenn sich der Mauszeiger auf dem Taster befindet und die linke Maustaste gedrückt wird. Die Methode `mMouseUpAction()` wird aufgerufen.
- ☞ Der Taster wird wieder in die Ausgangsposition bewegt, wenn die linke Maustaste losgelassen wird. Dabei ist es egal, ob sich der Mauszeiger noch auf dem Taster befindet oder nicht. Die Methode `mMouseUpAction()` wird aufgerufen.

Der Entwickler eines virtuellen Labors kann bei Verwendung dieses Bausteines für jeden Taster festlegen, welche Aktion bei einem Drücken oder / und Loslassen der Maustaste augeführt und an welcher Position der Taster verschoben werden soll. Der Translationsvektor kann durch den konkreten View als Parameter an den „Taster“-Controller (`3DCTRLTaster`) weitergegeben werden. Der Entwickler kann sowohl beim Eindrücken des Schalters als auch beim Loslassen eine Aktion durchführen. Sie können in den beiden Methoden `mMouseUpAction()` und `mMouseUpAction()`, die in der Klasse `3DCTRLTaster` abstrakt definiert sind, spezifiziert werden. Viele Taster führen die gleiche Aktion an ihren Geräten oder Behältern durch. Sie schalten sie ein oder aus, starten oder stoppen, öffnen oder schließen sie. Hierfür wurden entsprechende Controller entwickelt, die direkt genutzt werden können.

Drehschalter

Im Gegensatz zu Tastern erfolgt die Betätigung eines Drehschalters in diesem Visualisierungskonzept nicht durch das Drücken einer Maustaste, sondern durch eine Drag-Bewegung, die eine entsprechende Rotation des Schalters ausführt. Dabei können wie bei jedem View Rotationsgrenzen angegeben werden. Während der Drag-Bewegung

wird der View direkt manipuliert, d. h. die Mausbewegung des Anwenders wird in eine Rotation des 3D-Models umgesetzt. Anhand dieser Rotation meldet der Controller dem Modell den neuen Wert. Das kann auf zwei verschiedene Weisen durchgeführt werden. Entweder der Controller hat Kenntnisse über den Wertebereich des zu ändernden Zustands im Modell und kann die Berechnungen des absoluten Wertes selbst durchführen und setzen, oder der Controller übermittelt einen prozentualen Wert des Rotationsfortschrittes. Wurde der Schalter also um 50% gedreht, wird das Modell mit diesem Wert benachrichtigt. Im Modell muss dann eine Methode implementiert sein, die ein Setzen des Zustandes über einen prozentualen Wert erlaubt. Auf diese Weise braucht weder der View noch der Controller Kenntnisse über den Wertebereich, den der Zustand im Modell annehmen kann.

Ein Drehschalter wird nicht nur zum Einstellen eines Zustandes benutzt, sondern gleichzeitig zum Anzeigen des Modellzustandes. Neben der direkten Manipulation durch den Controller, kann die Ausrichtung des Drehschalters über eine Modelländerung verändert werden. Der View muss daher den Zustand des Modells überwachen und gegebenenfalls selbst eine Änderung seiner Ausrichtung durchführen.

Während eines Drags muss der View und der Modellstatus geändert werden. Aus diesem Grund wird bei jeder Drag-Bewegung vom `3DCtrl` die Methode `mWhileDraggedAction()` aufgerufen. Konkrete Controller können diese Methode überschreiben. Allgemeine Aufgaben eines Drehschalter wurden im Controller `3DCtrlSwitchRotate` implementiert. Er belegt die Rotations / Drag-Auswahlmodus sowie die Achse um die bei einer Mausbewegung gedreht wird, neu.

Kippschalter

Tasten ändern bei Betätigung ihre Position. Kippschalter hingegen werden um einen bestimmen Winkel gedreht. Es gibt verschiedene Ausprägungen: Kippschalter die nach Betätigung in ihre Ursprungsausrichtung zurückspringen oder solange in der gekippten Orientierung verharren bis der Schalter erneut betätigt wird. Der Entwickler kann analog zu den Tasten-Bausteinen festlegen, wann eine Aktion ausgeführt und wie der Kippschalter gedreht werden soll.

5.6.2 Dynamische Anzeigen

Viele Laborkomponenten in naturwissenschaftlich-technischen Laboren besitzen digitale Anzeigen, um dem Anwender einen Zustand der Komponente zu verdeutlichen. Im GENLAB-Framework werden Anzeigen durch die Verwendung von Textcastmember realisiert. Die Information wird als Text, also mit einem Font, auf die Laborkomponente platziert. Die Formatierung und Aktualisierung der Anzeige (Schriftart, -größe, -farbe etc.) kann über diesen Textcastmember erfolgen. 3D DREAMS untersützt die Verwendung von Textcastmember nicht. Die Modellierung der Buchstaben als 3D-Modelle wäre viel zu aufwendig, bzw. die dynamische Anpassung in einigen Fällen viel zu langsam. Eine flexible Lösung bietet die Möglichkeit der dynamischen

Zuweisung von Texturen auf 3D-Modellen in Kombination mit der Eigenschaft von DIRECTOR, Textcastmember in Bitmapcastmember umwandeln zu können. Die Vorgehensweise kann wie folgt näher beschrieben werden:

① Erstellung eines 3D-Modells, dass als Anzeige dient und die Anzeigentextur aufnimmt. Es kann einer Laborkomponente, beispielsweise einer Mikrowelle untergeordnet werden.

② Erstellung eines Textcastmembers mit den gewünchten Formatierungen

③ Schreiben der Information in den Textcastmember

④ Umwandeln des Textcastmember in einen Bitmapcastmember

⑤ Zuweisung des Bitmapcastmember als Textur auf das Anzeigen-3D-Modell

Bei einer Aktualisierung der Anzeige müssen jeweils die Schritte 3, 4 und 5 durchgeführt werden.

Da die meisten Anzeigen planar sind, wurde ein planares 3D-Modell erstellt, das in seiner Größe durch den Entwickler des Labors angepasst werden kann. Für die Erzeugung wird die Klasse `Anzeige3DView` verwendet. Sie bietet die grundlegenden Funktionen zum Aktualisieren der Anzeige. Ein konkretes Format der Anzeige kann dem Anzeigen-Objekt beim Erzeugen nicht als Parameter mitgegeben werden (s. Abschnitt 5.2). Daher wird die Formatzuweisung erst in einer Initialisierungsmethode vorgenommen. Eigens für die Formatierung wurde die Klasse `3DViewAnzeigeText` implementiert. In ihr werden grundlegende Methoden zur Formatierung angeboten, die in Unterklassen konkretisiert werden können. Neben den reinen Formatieraufgaben wird in ihnen die Zustandsüberwachung des jeweiligen Modells vorgenommen, um die konkrete Information für die Anzeige bereitzustellen. Anhand der Digitalanzeige der Mikrowelle kann das Zusammenspiel der Klassen verdeutlicht werden:

☞ Der View der Mikrowelle (`Mkr3DView`) erzeugt ein neues Format. In diesem Format sind konkrete Darstellungsparameter (Schriftart, Farbe, Größe der Anzeige etc.) spezifiziert. Außerdem überwacht dieses neu erzeugte Objekt das Modell der Mikrowelle auf evtl. Zustandsänderungen, die angezeigt werden müssen.

☞ Der Mikrowellen-View erzeugt ebenfalls die Anzeige (`Anzeige3DView`) und initialisiert sie mit dem Formatobjekt. Die Anzeige wird an die Mikrowelle attached, ausgerichtet und positioniert.

☞ Bei einer Zustandsänderung des Mikrowellen-Modells wird das Formatobjekt benachrichtigt. Dieses übergibt die neue Information an das Anzeigenobjekt (Instanz von `Anzeige3DView`)

☞ Das Anzeigenobjekt schreibt den Text in einen Textcastmember, erzeugt einen Bitmapcastmember daraus und weist den Bitmaptextcastmember als Textur dem Anzeigen 3D-Modell zu.

5.6.3 Virtueller Arbeitsbereich

Als eigenständiger Baustein, der mehrfach in Laboren verwendet werden kann, aber nicht in andere Laborkomponenten integriert wird, ist der *Virtuelle Arbeitsbereich*. Durch eine uneingeschränkte Bewegung oder Rotation können 3D-Objekte außerhalb des Sichtfeldes des Anwenders geraten und unter Umständen nicht mehr in den Sichtbereich zurückgestellt werden. Dieses Problem lässt sich sich in einer 3D-Darstellung beheben, indem ein virtueller, nicht sichtbarer Raum in der Größe des gewünschten Bewegungsspielraums aufgespannt wird. Die Kollisionserkennung verhindert das Bewegen oder Rotieren eines 3D-Objektes durch Wand, Boden oder Decke des unsichtbaren Raums. Auf diese Weise kann nicht nur vermieden werden, dass ein 3D-Objekt das Sichtfeld verlässt, sondern auch dass es nicht beliebig in die Tiefe hinein oder zu nah an die Kamera bewegt wird. Zur Aufspannung eines solchen Raumes wurde die Klasse `virtRoom3DView` implementiert. Sie wird von der Klasse `3DViewObjects` abgeleitet und aggregiert sechs Viewobjekte, die den Raum definieren (Decke, Boden, Vorderwand, Rückwand, linke Seitenwand und rechte Seitenwand). Es gibt keine Nutzer-Events, die diese Objekte empfangen können. Allerdings werden sie bei der Kollisionserkennung berücksichtigt, so dass Objekte, die einen dieser Views berühren, nicht weiter hinaus verschoben werden können. Die Begrenzungen werden in einer von der Klasse `virtRoom3DView` abgeleiteten Klasse vom Entwickler so platziert, dass sie genau den gewünschten Arbeitsraum definieren. Außerdem gibt es die Möglichkeit, den virtuellen Begrenzungsraum ganz oder teilweise zu aktivieren und deaktivieren.

5.7 Performancesteigernde Maßnahmen

Jede Transformation eines 3D-Objektes kann auch Auswirkungen auf das Aussehen anderer 3D-Objekte haben. Teile anderer Objekte können verdeckt oder wieder aufgedeckt werden. 3D DREAMS rendert also nicht nur die 3D-Objekte neu, die direkt transformiert werden, sondern auch diejenigen, die indirekt von dieser Transformation betroffen sind. Dies hat zur Folge das 3D-Objekte wie Arbeitsflächen und Wände bei nahezu jeder Objektänderung erneut komplett gerendert werden, wodurch auf Standard Mulitmedia-PCs die Performance einer Animation auf weit unter 15fps (Frames per Second) sinkt. Dies erweckt beim Anwender nicht mehr den Eindruck einer flüssigen Bewegung. Aus diesem Grund wurden zwei Möglichkeiten zur Performancesteigerung implementiert.

5.7.1 Darstellung von Arbeitsflächen und Wänden

In 3D DREAMS gibt es die Möglichkeit, die Sichtbarkeit der 3D-Objekte auszuschalten. Sie werden zwar bei der Kollisionserkennung weiterhin berücksichtigt, brauchen jedoch nicht gezeichnet werden. Diese Eigenschaft wurde bereits beim virtuellen Arbeitsbereich (Abschnitt 5.6.3) ausgenutzt. Ein Geschwindikgkeitsgewinn kann daher

schon erzielt werden, indem Wände und Arbeitsflächen, als unsichtbar dargestellt werden.

Damit trotzdem auch visuell die Arbeitsfläche oder Wand für den Anwender vorhanden ist, kann der Viewport transparent dargestellt und hinter dem Viewport ein DIRECTOR-Sprite mit einer Bitmap platziert werden, die den Eindruck einer Arbeitsumgebung vermittelt. Diese Technik wird ebenfalls in GENLAB verwendet. Sie ist nur bei starren Kamerapositionen und -ausrichtungen einsetzbar. Beim Wechseln eines Arbeitsplatzes bzw. einer Kamera müssen die Bitmaps ausgetauscht werden, wenn die Arbeitsplätze unterschiedlich aussehen sollen.

Abbildung 5.6 zeigt die Simulation einer 3D-Arbeitsumgebung über eine Hintergrundbitmap hinter einem transparenten Viewport.

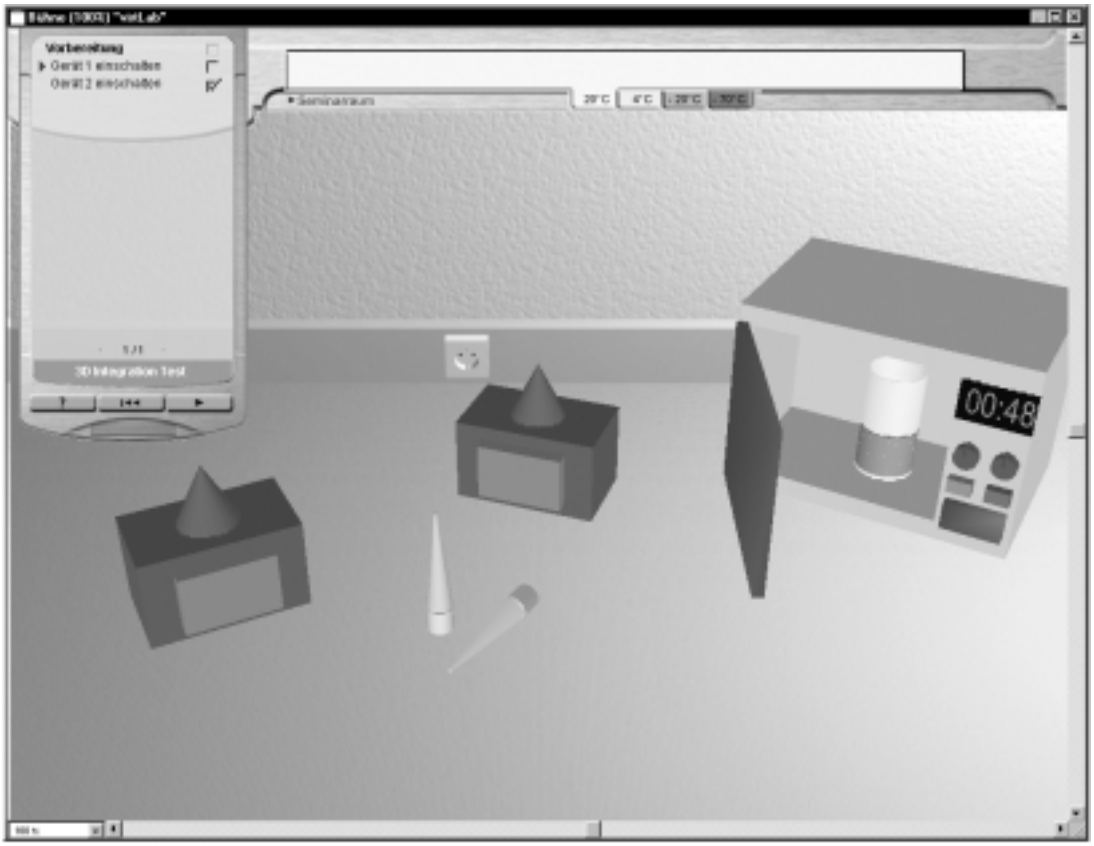

Abbildung 5.6: Verwendung einer Hintergrundbitmap, um den Eindruck einer Arbeitsumgebung zu vermitteln

5.7.2 Wireframedarstellung beim Bewegen von 3D-Objekten

Einen geringen Geschwindigkeitsgewinn von ungefähr 3fps bei der Animation von Objekten kann erhalten werden, wenn der Schattierungsmodus ausgeschaltet wird und

die betroffenen Objekte im Wireframe-Modus dargestellt werden. 3D DREAMS bietet die Möglichkeit, für jedes Objekt einzeln zu bestimmen, in welchem Schattierungsmodus es dargestellt werden soll. Es wurde testweise ein Funktion implementiert, die beim Draggen den Wireframe-Modus des bewegten Objektes einschaltet und beim Droppen wieder in den ursprünglichen Schattierungsmodus zurückschaltet. Zur (De-)Aktivierung dieser Optimierung muss die globale Methode `mSetOptimize([TRUE|FALSE])` mit dem entsprechenden booleschen Wert aufgerufen werden. In Abbildung 5.7 ist die Mikrowelle mit Inhalt beim Draggen im Wireframe-Modus dargestellt.

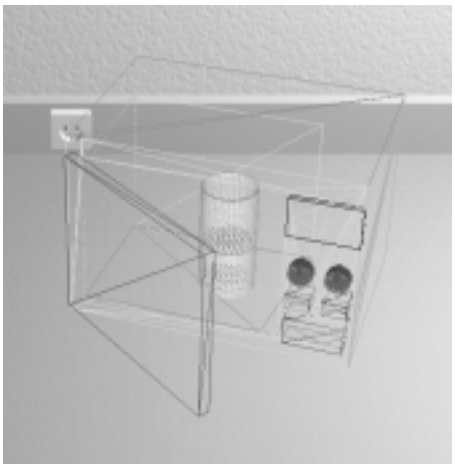

Abbildung 5.7: Wireframe-Modus beim Drag der Mikrowelle

5.8 Erweiterung und Korrekturen von 3D DREAMS

In diesem Abschnitt werden Implementationen beschrieben, die einige Unzulänglichkeiten der eingesetzten 3D DREAMS - Version beheben. Bei Verwendung neuerer Versionen von 3D DREAMS oder anderer Render-Engines und Kollisionserkennungsverfahren können diese Implementationen keine Bedeutung mehr haben.

5.8.1 Kollisionen, Objektannäherung

In dieser Arbeit wurde die Kollisionserkennung von 3D DREAMS verwendet. Wird eine Kollision zwischen zwei Objekten festgestellt, wird die letzte Bewegung bzw. Rotation, die diese Kollision verursachen würde, nicht mehr ausgeführt. Dies führt dazu, dass eine Kollision gemeldet wird, obwohl die Objekte für den Nutzer noch weit von einander entfernt erscheinen. Dieser Umstand wird besonders deutlich, wenn der Nutzer ein Objekt mit schneller Mausbewegung positionieren will. Denn dann sind die Distanzen, die das Objekt innerhalb eines Zyklusses zurücklegen muss groß. Um dieses Problem zu lösen, wurde die Methode `mImproveLoc()` in der Viewklasse `3DViewDrag` implementiert. Sie wird aufgerufen, wenn eine Kollision zustandekommt und platziert das aktive Kollisionsobjekt möglichst nah an das passive Kollisionsobjekt heran. Der Algorithmus dieser Methode ist vergleichbar mit der binären Suche in einer Liste nach Ottmann und Widmayer[OW90] und wird rekursiv beschrieben. Da die Anzahl der zu überprüfenden Positionen unendlich groß ist und der Algorithmus daher unendlich weiterrechnen würde, kann die Rekursionstiefe angegeben werden. Alternativ könnte auch die Differenz zwischen alter und neuer Position als Abbruchbedingung definiert werden.

```
on mImproveLoc me, dLoc, dTimes
  -- dLoc = gewuenschte Zielposition
  -- dTimes = Rekursionstiefe

  if voidP(dTimes) then
    dTimes = 10
  end if

  -- aktuelle Position speichern
  toldPos = me.mGetLoc()

  -- Differenz zwischen gewuenschter Position
  -- und aktueller Position berechnen
  tDiff = dLoc - tOldPos

  -- Differenz halbieren
  tDiff = tDiff / 2
```

```
  -- mögliche neue Position auf halber
  -- Strecke berechnen
  tNewPos = tOldpos + tDiff

  -- Objekt auf neue Position setzen
  me.mSetLoc(tNewPos)

  -- Rekursionsabbruchbedingung
  if dTimes = 0 then
    exit
  end if

  -- Alte und neue Position vergleichen.
  -- Sind beide Positionen gleich, dann führt die
  -- Positionierung auf halbe Strecke ebenfalls
  -- zu einer Kollision, d. h. der Algorithmus muss
  -- mit dem kleineren Bereich fortgesetzt werden.
  -- Andernfalls wurde die Positionierung erfolgreich
  -- durchgeführt. Der "hintere" Bereich genauer
  -- untersucht werden.
  if tOldPos = me.mGetLoc() then
    me.mImproveLoc(tNewPos, dTimes-1)
  else
    me.mImproveLoc(dLoc, dTimes-1)
  end if
end
```

Die Implementierung einer Methode zur Annäherung an ein Objekt durch Rotation kann analog implementiert werden.

5.8.2 Erzeugung von Objekten in der Runtime-Umgebung

Beim Erzeugen neuer Objekte in einem Projektor werden den 3D-Objekten keine Materialdaten zugewiesen. Die Geometrie wird korrekt dargestellt. Zur Lösung dieses Problems wurden die Methoden `mSavematerialData()` und `mRestorematerialData()` in der Klasse `3dViewObjects` implementiert. Die Routine `mSaveMaterialData()` liest im *Authoring Mode* die Materialdaten über die Schnittstellenmethode `mGetMaterial()` aus und schreibt die Daten in einen Castmember. Im *Runtime Mode* liest die Methode `mRestoreMaterialData()` bei Initialisierung der 3D-Objekte die Materialdaten wieder aus dem Castmember und setzt sie mittels der Methode `mSetMaterial()`.

5.9 Zusammenfassung

In diesem Kapitel wurden die für die Erstellung virtueller dreidimensionaler Labore notwendigen Basisklassen und Konzepte aus dem Entwurf implementiert. Das GENLAB-Framework wurde um die Möglichkeit erweitert, Versuche mit dreidimensionalen Objekten zu entwickeln. Diese können direkt ohne den aufwendigen Arbeitsablauf der zweidimensionalen View-Erstellung in das Autorensystem integriert werden. Die in diesem Kapitel entwickelten Algorithmen und Datenstrukturen erlauben eine dynamische Erzeugung dreidimensionaler Objekte, die durch den Anwender interaktiv transformiert werden können. Der Entwickler eines virtuellen Labors kann für jedes 3D-Objekt entscheiden, welche Transformationsmethoden erlaubt sind, welche Einschränkungen es gibt und wie diese vom Anwender am einfachsten durchgeführt werden können. Weiterhin wurde die Möglichkeit implementiert, dass 3D-Objekte untereinander auf verschiedene Weise miteinander interagieren können. Neben der „echten“ Interaktion, bei der die 3D-Objekte ihre Modelleigenschaften gegenseitig beeinflussen können, exisitert eine rein auf den View bezogene Interaktion, die es ermöglicht 3D-Objekte auf anderen abzustellen. Dank der Verwendung des MVC-Konzeptes sind die schon implementierten Modelle des GENLAB-Frameworks direkt nutzbar. Für häufig vorkommende Elemente in bzw. auf Geräten wurden exemplarisch Bausteine erstellt, die die Entwicklung virtueller Labore erleichtern. Die Implementierung performancesteigernde Maßnahmen (unsichtbare Wände und Arbeitsflächen, sowie eine Wireframedarstellung beim Bewegen eines Objektes) ermöglichen ein flüssiges, interaktives Arbeiten mit 3D-Objekten auf üblichen Multimedia-PCs.

6 Erstellung eines prototypischen Labors

Dieses Kapitel zeigt in Form eines Entwicklerleitfadens die prototypische Erstellung eines virtuellen Labors unter Verwendung des GENLAB-Frameworks und der Erweiterung zur Integration dreidimensionaler Modelle. Der Workflow zur Erstellung eines solchen Labors wird in Abbildung 6.1 beschrieben. Dabei bleiben die grau hinterlegten Kästen unberührt von der 3D-Visualisierung und werden hier nicht weiter erörtert. Der Workflow ist nach den an der Erstellung einer virtuellen Labors beteiligten Personen (siehe Abschnitt 2.1) gegliedert. Im Vordergrund dieses Kapitels steht die Entwicklung einer virtuellen Versuchsumgebung. Die Erstellung einer Rahmenhandlung wird nicht beschrieben, da sie keine neuen Aspekte bzgl. der 3D-Integration hinzufügt.

Nach einigen Installationshinweisen (Abschnitt 6.1) sollen insbesondere die Vorgehensweise bei der Konstruktion der 3D-Modelle für interaktive Umgebungen (Abschnitt 6.2) detailliert betrachtet werden. In einem daran anschließenden Abschnitt (6.3) wird der Import der 3D-Modelle erläutert. Es folgt eine Beschreibung zur Erstellung der notwendigen View- und Controllerklassen durch den Informatiker. Die Modelklassen brauchen nicht berücksichtigt zu werden, da das 3D-Visualisierungskonzept so entworfen wurde, dass bereits bestehende GENLAB-Modelle direkt genutzt werden könnnen. Eine weitere Aufgabe, die sich Informatiker und Multimedia-Designer teilen, ist die Anbindung der Viewklassen an die 3D-Modelle. Sie wird in Abschnitt 6.5 erläutert. Nachdem die Szene korrekt konfiguriert (s. Abschnitt 6.6), d. h. Kameras, Arbeitsbereiche und Räume angelegt wurden und die Versuchsabläufe der Fachkdidaktiker in die Versuchssteuerung integriert sind, kann ein neues Module definiert werden. Eine Beispieldefinition wird in Abschnitt 6.7 gezeigt. Im Abschluss werden notwendige Maßnahmen für eine Distribution des virtuellen Labors dargelegt.

6.1 Installationshinweise / Vorbereitung

Grundlage für dieses Kapitel ist eine funktionierende Installation von DIRECTOR mit dem XTRA 3D DREAMS, dem GENLAB-Framework mit den in dieser Arbeit entwickelten Erweiterungen und einem 3D-Grafikprogramm, welches das binäre 3D-Sudio Format unterstützt. Bei der Installation von 3D DREAMS wird ein XTRA in alle auf dem Rechner vorliegenden DIRECTOR-Versionen in das XTRA-Verzeichnis kopiert. Ein weiteres 3D DREAMS - XTRA befindet sich im Installationsverzeichnis

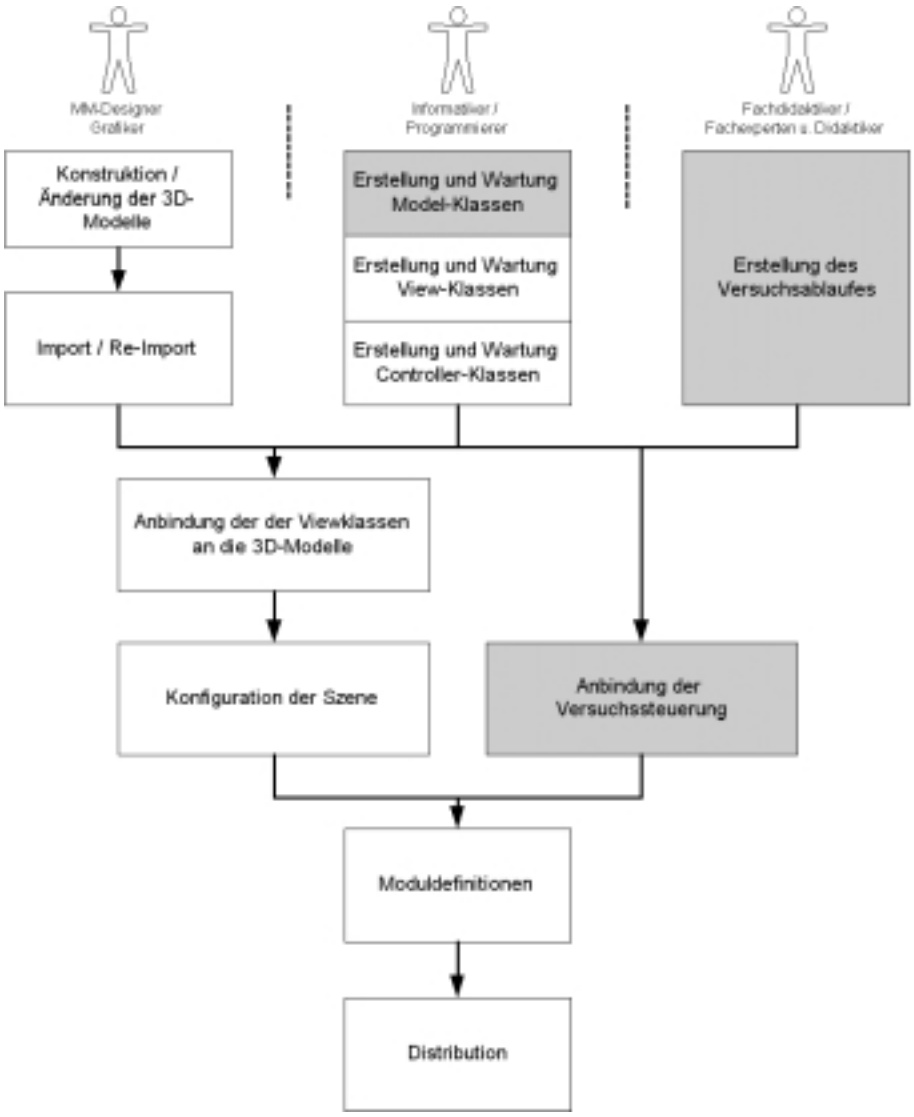

Abbildung 6.1: Workflow zur Erstellung eines Versuches mit 3D-Integration

von 3D DREAMS. Es wird bei einer Distribution des Labors benötigt (s. Abschnitt 6.8).

6.2 Konstruktion der 3D-Modelle

Die Verwendung von 3D-Modellen in einer interaktiven Laborumgebung erfordert eine besondere Vorgehensweise bei der Konstruktion der 3D-Modelle. Sie wird anhand der Modellierung einer Mikrowelle in dem 3D-Grafikprogramm CINEMA4D beschrieben. Es werden jedoch keine speziellen Konstruktionsmethoden verwendet, so dass auch andere 3D-Grafikprogramme verwendet werden können.

6.2.1 Anforderungen an 3D-Modelle für den interaktiven Gebrauch

Bei der Konstruktion interaktiver 3D-Modelle sollte immer darauf geachtet werden, dass die Polygonzahl der Modelle und die Anzahl der „Unterobjekte“ möglichst gering ist. Je geringer die Anzahl der Polygone eines Objektes ist, desto kürzer ist die Berechnungs- und Darstellungszeit (s. Abschnitt 2.2.1). Bei kantigen Objekten ist es einfach, die Polygonzahl gering zu halten. Runde Objekte benötigen für eine realitätsnahe Darstellung wesentlich mehr Polygone, damit die kantigen Übergänge zwischen den Polygonen nicht mehr sichtbar sind. Ab einer bestimmten Polygonanzahl wird jedoch die grafische Qualität nicht mehr verbessert (s. Abschnitt 2.4.2). Viele 3D-Grafikprogramme besitzen eine Funktion zum Optimieren der Polygonzahl. Bei Tests in CINEMA 4D wurde dabei die Polygonzahl jedoch nur um wenige Prozente verringert. Ist eine solche Funktion nicht vorhanden, kann auf externe Programme zur Polygonreduktion zurückgegriffen werden. [Bou97] gibt weitere Hinweise zur Polygonreduktion. Am sinnvollsten wird jeodoch schon während der Konstruktion auf eine geringe Polygonanzahl geachtet, anstatt sie im nachhinein zu verringern.

Die Zahl der Unterobjekte sollte deswegen klein gehalten werden, weil jedes einzelne Objekt später in die interaktive 3D-Szene importiert und durch die Render-Engine verwaltet werden muss. Je umfangreicher die Szene wird, desto aufwendiger ist diese Verwaltung. Beim Bewegen eines Objektes beispielsweise müssen alle Unterobjekte mitbewegt werden, d. h. die Szene muss von diesem Objekt bis zum letzten Kindobjekt durchlaufen und die Bewegungsinformation weitergegeben werden. Je weniger Unterobjekte für dieses Objekt benötigt werden, desto schneller kann die Szene durchlaufen werden. Folgende Regel kann zur Reduktion der Objektanzahl verwendet werden:

Zwei Objekte können genau dann zusammengefügt werden, wenn

① beide Objekte nicht einzeln interaktiv durch den Nuzter oder durch das Programm manipuliert werden müssen und

② beide Objekte die gleichen Material- und Textureigenschaften besitzen.

Oft entstehen im Konstruktionsprozess viele Objekte, die zu einem Objekt verbunden werden können. Beispielsweise konstruiert ein 3D-Grafikprogramm einen einfacher Zylinder durch die drei Einzelobjekte Grundkörper, obere und untere Deckfläche. Die meisten 3D-Programme bieten eine Funktion zum Verbinden dieser Objektteile zu einem Objekt. Dabei sollte beachtet werden, dass das *Gruppieren* zweier Objekte nichts mit dem *Verbinden* der Objekte gemeinsam hat. Eine Gruppe ist nur ein Zusammenhalt mehrerer Objekte innerhalb der 3D-Szene. *Verbinden* hingegen bedeutet das Zusammenführen der einzelnen Polygone der Objekte zu einem Objekt. Im Gegensatz zur Gruppierung kann eine Verbindung im virtuellen Labor also nicht mehr aufgehoben werden.

6.2.2 Modellierungstechniken

In Abschnitt 2.4 wurden allgemeine Hinweise zu den Modellierungstechniken gegeben. Hinsichtlich der Verwendung von 3D DREAMS als Render-Engine ist hinzuzufügen, dass das Modellieren in CINEMA 4D mit einer NURBS-Technik mit anschließendem Umwandeln in einen Flächenkörper keine guten Ergebnisse erzielt. Solche Objekte werden von 3D DREAMS oft fehlerbehaftet dargestellt. Dies wirkt sich durch teilweise fehlende Polygone aus. Empfehlenswerter ist die Modellierung über vordefinierte Primitive in Verbindung mit booleschen Operationen.

6.2.3 Konkrete Modellierung einer Mikrowelle

Am Beispiel einer Mikrowelle wird die Modellierung eines 3D-Modells zur späteren interaktiven Verwendung demonstriert. Die einzelnen Konstruktionsschritte sind in den Abbildungen 6.2 und 6.3 dargestellt.

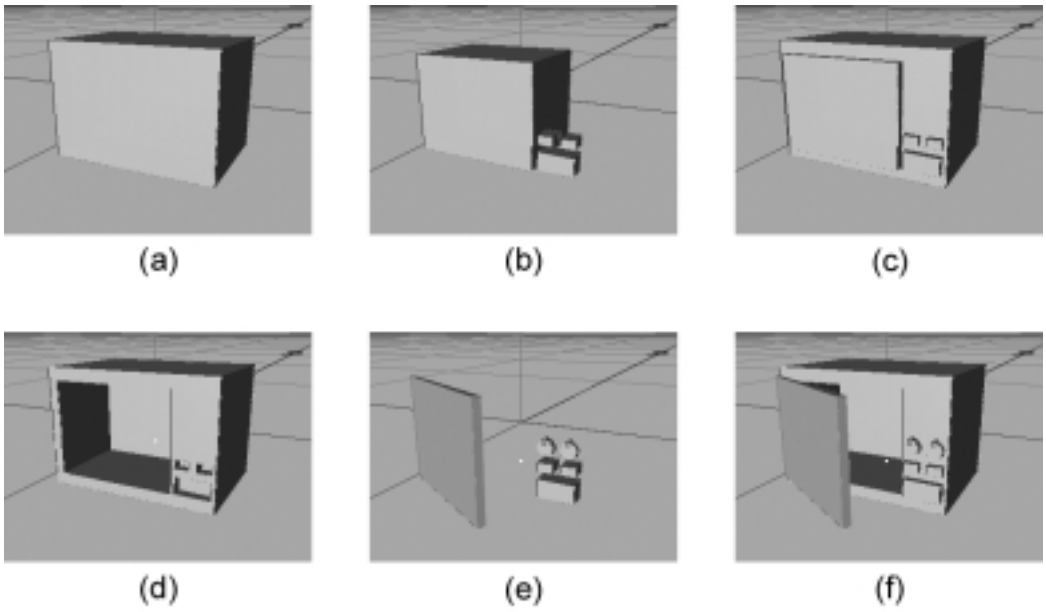

Abbildung 6.2: Die Konstruktion eines 3D-Modells am Beispiel einer Mikrowelle

Folgende Konstruktionsschritte sind auszuführen:

Abb. 6.2 (a) Erstellung des Grundkörpers

Unter *Grundkörper* ist hier das Gerüst zu verstehen, das alle anderen Elemente (Tür, Schalter, etc.) trägt. Bei der Modellierung sollte schon auf geeignete Maße geachtet werden, da innerhalb des Labors häufig mit konkreten Größen- und Distanzangaben gearbeitet wird. Vorzugsweise werden also für die Konstruktion die realen Grundmaße einer Mikrowelle genommen. Der erste Schritt im Falle der Mikrowelle ist die Erstellung eines Quaders in der Größe der Mikrowelle.

Abb. 6.2 (b) & (c) Erstellung der Objekte für Hohlräume

Als nächstes müssen die Hohlräume für Schalter und Aufnahmebereiche erstellt werden. Für die Generierung der Hohlräume wird eine boolesche Operation verwendet. Deshalb werden Objekte in der Größe der Hohlräume zur Subtraktion benötigt und an die korrekten Stellen positioniert. Auch bei diesen Objekten sollte darauf geachtet werden, dass die Polygonzahl gering ist. Einige von ihnen können später als interaktive Elemente weiterverwendet werden.

Abb. 6.2 (d) Boolesche Operation (Subtraktion)

Die Elemente für den Hohlraum werden vom Quader abgezogen. Der Mikrowellengrundkörper entsteht. Er wird in der interaktiven Umgebung das in der Szenen-Hierarchie oberste Objekt der Mikrowelle darstellen.

Abb. 6.2 (e) & (f) Erstellung der interaktiven Elemente

In einem weiteren Schritt werden alle Elemente erstellt, die entweder interaktiv durch den Nutzer manipuliert oder durch das Programm referenziert werden müssen (Abb. 6.2 (e)). In diesem speziellen Fall können der Start-, Stop- und Türknopf aus den Objekten für die boolesche Subtraktion übernommen werden. Die Tür, das Watt- und Zeitrad müssen modelliert werden. Alle Objekte werden an die entsprechenden Positionen gesetzt (Abb. 6.2 (f)) und in der Szenenhierarchie unterhalb des Mikrowellen-Grundkörpers platziert.

Abb. 6.3 (a) & (b) Reduktion der Objekte

Wird das Wattrad der Mikrowelle näher betrachtet, fällt auf, dass es aus zwei Elementen zusammengesetzt ist, nämlich aus dem Zeiger und dem Wattrad selbst (Abb. 6.3 (a)). Sie werden im Labor jedoch nie getrennt behandelt und sind aus dem gleichen Material. Sie können deshalb verbunden werden (Abb. 6.3 (b)). Analog ist mit dem Zeitrad zu verfahren.

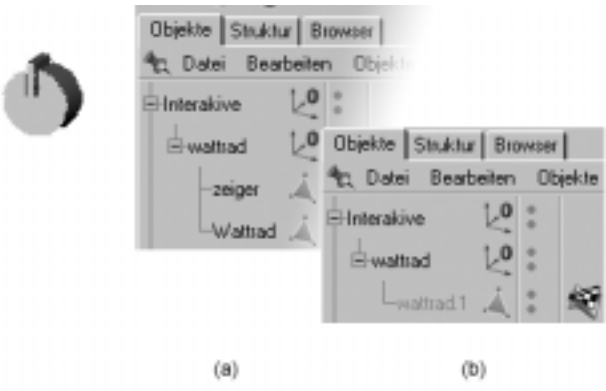

Abbildung 6.3: Verbinden zweier 3D-Objekte

Die Szenenhierarchie der Mikrowelle ist in Abbildung 6.4 dargestellt. Sie ist sehr übersichtlich und enthält nur die nötigsten Elemente.

Gegebenenfalls kann nun die Polygonanzahl optimiert werden. CINEMA 4D leistet bei der Optimierung eine Polygonreduktion von 1248 auf 1183 Polygone.

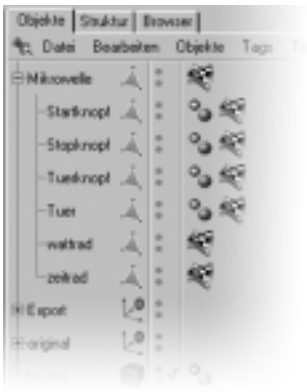

Abbildung 6.4: Hierarchie des Mikrowellen-Modells

Nach der Modellierung können die einzelnen Elemente mit Material- und Texturdaten versehen werden. Beim Rendern der Objekte von 3D DREAMS werden die Farbwerte und der Transparenzfaktor berücksichtigt. Für Texturen sieht 3D DREAMS eine planare Projektion vor, was bedeutet, dass eine Textur jeweils komplett über das Element gemapped wird.

Nach Abschluss der Modellierung muss die Datei im 3DS-Format abgespeichert werden.

6.3 Import der 3D-Modelle

Der Import der 3D-Modelle kann erfolgen, wenn bereits ein Viewport erstellt wurde. Durch Angabe des Dateinamens der 3D-Szenendatei, werden die in dieser Datei gespeicherten 3D-Objekte angezeigt. Die Hierarchie wird jedoch nicht mehr korrekt dargestellt, alle Objekte befinden sich auf der gleichen Hierarchieebene. Der Entwickler kann nun die zu importierenden Objekte auswählen. Am Beispiel der Mikrowelle entsteht die in Abbildung 6.5 gezeigte 3D-Szene.

Der Entwickler kann jetzt die Hierarchie, die Position und Ausrichtung wie gewünscht anpassen. In diesem Beispiel wird sie wie in Abbildung 6.6 gezeigt angeordnet.

Der Mikrowellengrundkörper dient als Vaterobjekt für die Unterobjekte Zeitrad, Wattrad, Tuer, Tuerknopf, Stopknopf und Startknopf. Die in der Abbildung zu sehenen Positionen sind globale Koordinaten. Die Objekte werden vorerst in einen für den Anwender nicht sichtbaren Bereich gestellt, damit durch das Programm Manipulationen an den 3D-Objekten vorgenommern werden können, die vom Anwender nicht gesehen werden sollen.

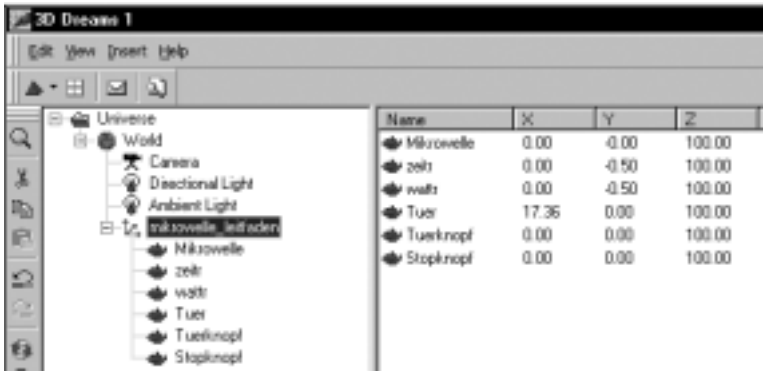

Abbildung 6.5: Hierarchie der Mikrowelle nach dem Import in 3D DREAMS

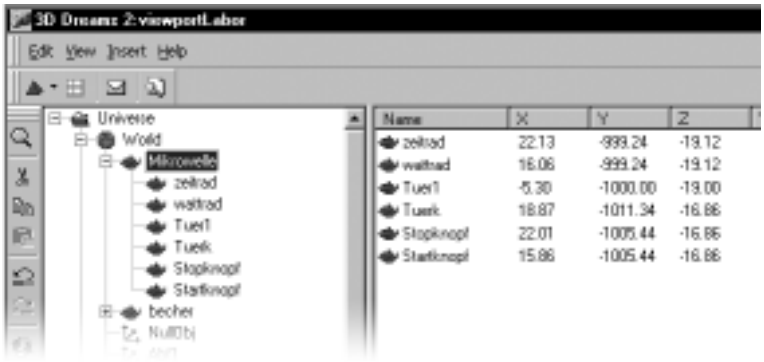

Abbildung 6.6: Geänderte Hierarchie der Mikrowelle in 3D DREAMS

6.4 Erstellen der View und Controllerklassen

Die Erstellung der View- und Controllerklassen durch den Informatiker wird ebenfalls an einigen Beispielen für die Mikrowelle gezeigt. Es wird mit der Implementierung der Viewklasse für den Mikrowellen-Grundkörper begonnen. Sie wird `Mkr3DView` genannt. Die Klasse leitet sich von der Klasse `3DViewObjectsRepository` ab, weil die Mikrowelle in der Lage ist, andere Objekte aufzunehmen. Die Anzahl der Objekte, die die Mikrowelle aufnehmen kann, wird durch das Modell der Mikrowelle bestimmt. Als Modell wird die Klasse `classModelMikrowelle` aus GENLAB verwendet. Es definiert eine Aufnahmekapazität von maximal zwei Objekten. In der Viewklasse müssen die Positionen für die evtl. aufgenommenen Objekte definiert werden. Zusätzlich muss die Position für ein herausgenommenes Objekt festgelegt werden. Weiterhin wird jedem Viewobjekt ein eindeutiger Name zugewiesen. Dieser Name und ein automatisch generierter Schlüssel bilden den Castmembernamen, unter welchem die Materialdaten abgespeichert werden. Die Methode `new()` der Klasse `Mkr3DView` sieht folgendermaßen aus:

```
on new me
  tOffsets = [ [#x:0, #y:-4, #z:-5],[#x:-15, #y:-4, #z:-5]]
  tRemoveOffset = [#x:0, #y:10, z:-30]
```

```
  me.ancestor=new(script "3DViewObjectsRepository", tOffsets,\
                                                    tRemoveOffset)
  me.pViewName = "Mkr3DView"
  return me
end
```

In der `mInit()`-Routine der Mikrowelle müssen folgende Operationen durchgeführt werden:

① Erzeugung des Models und Speicherung dessen Referenz

② Erzeugung des Controllers und Speicherung dessen Referenz

③ Festlegen eventueller Rotations- und Bewegungseinschränkungen

④ Initialisierung des Materials (wg. des Problems, dass die Materialien in einer Runtime-Version nicht zugewiesen werden)

⑤ Erzeugung und Positionierung einer Zeitanzeige und deren Format.

⑥ Initialisierung des Modells

⑦ Definition der aufnehmbaren Objekte

Dem View der Mikrowelle wird das Modell `classModelMikrowelle` und ein einfacher Controller (`3DCTRLMoveRotate`) zugeordnet, damit sie rotiert und bewegt werden kann. Der Rotationsbereich wird eingeschränkt. Sie lässt sich nur um die Y-Achse drehen. Ein planares Display für die Zeitanzeige und ein speziell für diese Anzeige implementiertes Format (`3DViewAnzeigeTextMkr`) wird erzeugt. Das Anzeigeobjekt wird mit dem Formatobjekt konfiguriert, an die Mikrowelle *attached* und ausgerichtet, damit es für den Anwender den Eindruck erweckt, als sei sie ein fester Bestandteil der Mikrowelle. Als letztes wird das Modell initialisiert und festgelegt, welche Objektarten in die Mikrowelle gestellt werden dürfen. Dazu gehören hier das Reaktionsgefäß und der Messbecher. Die `mInit()`-Methode der Klasse `Mkr3DView` sieht wie folgt aus:

```
on mInit me
  me.mSetModel(new(script "classModelMikrowelle"))
  me.pRotateConstraints = [#yaw:[], #pitch:[0,0], #roll:[0,0]]
  me.mSetController(new(script "3DCTRLMoveRotate",\
                                                 me, me.pModel))
  me.mInitMaterial()

  -- neue Anzeige erzeugen
  me.pAnzeigeView = anzeigePlanarMaster2.mNew()

  -- Format fuer die Anzeige erzeugen
  tFormat = new(script "3DViewAnzeigeTextMkr",\
                                      me.pAnzeigeView, me.pModel)
```

```
  -- Anzeige initalisieren
  me.pAnzeigeView.mInit(tFormat, "MkrAnzeige")

  -- die childeren initialisieren
  me.mInitChildren()

  -- die Anzeige ausrichten
  me.pAnzeigeView.mAttachMe()
  tLoc = [#x: 19.0, #y: 10.0, #z: -17.7]  --box
  me.pAnzeigeView.mSetSolidness(FALSE)
  me.pAnzeigeView.mSetLoc(tLoc)
  me.pAnzeigeView.mSetOrientation([#yaw:0.0, #pitch:0.0, #roll:0.0])

  -- das Modell initialisieren
  me.mGetModel().mInit()
  me.mSetSolidness()

  -- Objekte, die in die Mkr gestellt werden koennen:
  me.mAddToDropTable("Reaktionsgefäß",#repository)
  me.mAddToDropTable("Meßbecher",#repository)
end
```

Eine weitere Methode (`mHandleDropObject()`) behandelt die auf die Mikrowelle gedroppten Objekte. Falls das gedroppte Objekt keine anderen Objekte trägt wird im Fall der Mikrowelle die Einfüge-Routine des Vorfahren aufgerufen.

```
on mHandleDropObject me
  if me.pInteractionObj.pCarrierList.count = 0 then
    me.mEinfuegen(me.pInteractionObj)
  else
    me.mFehler("Auf dem Objekt befinden sich weitere Objekte.\
         Dies ist für die Einfüge-Operation nicht zulässig" )
  end if
end
```

Die Methoden zum Hilighting und Unhilighting werden in dem Mikrowellen-View überschrieben.

Üblicherweise werden für jedes einzelne Element eigene View-Skripte implementiert. Die meisten sind jedoch wesentlich einfacher als die Klasse `Mkr3DView`, wie beispielsweise das Skript `MkrStart3DView` für den Startknopf der Mikrowelle zeigt:

```
on new me
  me.ancestor=new(script "3DViewDrag")
  me.pViewName = "MkrStart"
  return me
end
```

```
on mInit me, dModel
  me.mSetModel(dModel)
  me.mSetController(new(script "3DCTRLTasterStart", 2.5,\
                                                me, me.pModel))
  me.mInitMaterial()
  me.mSetSolidness()
end
```

Eine weitere Klasse wird hier kurz vorgestellt, damit die Kommunikation zwischen Model und View, sowie die Verwendung von Transformationsknoten deutlich wird. Dazu wird das Skript `MkrTuer3DView` näher betrachtet. Es stellt den View für die Tür der Mikrowelle dar:

```
01  on new me
02    me.ancestor=new(script "3DViewDrag")
03    me.pViewName = "MkrTuer3DView"
04    return me
05  end

06  on mInit me, dModel
07    me.mSetModel(dModel)
08    me.pRotateConstraints = [#yaw:[0,85], #pitch:[0,0],\
                               #roll:[0,0]]
09    tPos = [#x:-18.5, #y:0, #z:0]
10    tOrient = me.mGetOrientation()
11    me.mCreateTransformnode(tPos, tOrient)
12    me.pModel.mSubscribe(#Verschlossen, me)
13    me.mSetController(new(script "3DCTRLRotateMkrTuer",\
                          me, me.pModel, me.pRotateConstraints))
14    me.mInitMaterial()
15    me.mSetSolidness()
16  end

17  on mUpdate me, dProperty
18    case dProperty of
19      #Verschlossen:
20        tOrientation = me.mGetOrientation()
21        if not me.pModel.pVerschlossen then
22          tOrientation.yaw = 60.0
23          me.pCTRL.mResumeControl(#All)
24          me.mDetachMe()
25        else
26          tOrientation.yaw = 0.0
27          me.pCTRL.mHoldControl(#All)
28          me.mAttachMe()
29        end if
30        me.mSetOrientation(tOrientation)
31    end case
32  end
```

In Zeile 08 werden die Rotationswinkel festgelegt. Sie bedeuten, dass die Tür um maximal 85 Grad um die Y-Achse rotiert werden kann. Um andere Achsen kann nicht gedreht werden. Da sich die Tür jedoch nicht um ihren Mittelpunkt dreht, sondern um die linke Türangel, wird ein Transformationsknoten erzeugt, der sich am linken Rand der Tür befindet, um 18.5 Einheiten nach links verschoben (Zeilen 09, 10 und 11). Alle Rotationen werden über diesen Transformationsknoten vollzogen, so dass die Tür um ihre linke Türangel gedreht wird. Die Tür lässt sich jedoch nur drehen, wenn sie nicht verschlossen ist. Ansonsten muss erst der Türknopf betätigt werden, der die Tür in eine bestimmte Orientierung versetzt. Ein direktes Öffnen einer verschlossenen Tür ist nicht erlaubt. Zu diesem Zweck wird beim Modell die Eigenschaft `#Verschlossen` überwacht (Zeile 12). Ändert sich diese, ruft das Modell die Methode `mUpdate()` des Views auf. Ist die Mikrowelle nun geschlossen, wird die Tür auf einen Yaw-Winkel von 0 Grad gesetzt. Da sie nicht mehr rotiert werden darf, wird der entsprechende Controller angewiesen die gesamte Interaktion mit dem Nutzer temporär einzustellen (Zeile 27). Wird die Mikrowelle wieder geöffnet, sei es durch Betätigung des Türknopfes oder durch eine Manipulation des Modells, springt die Tür auf 60 Grad und der Controller muss seine Aktivität wieder aufnehmen (Zeile 23). Zeile 28 sorgt dafür, dass bei geschlossener Mikrowelle alle Nutzer-Events von der Tür an den Grundkörper der Mikrowelle (sprich `Mkr3DView`) weitergeleitet werden. Zeile 24 schaltet diesen Mechanismus wieder ab.

Neben der Erstellung der Viewklassen müssen eventuell weitere Controller-Klassen implementiert werden. Als Beispiel für einen speziellen Controller wird hier der Controller für die Mikrowellentür, die Klasse `3DCTRLRotateMkrTuer`, betrachtet. Er leitet sich von der Klasse `3DCTRLRotate` und besitzt dadurch die Eigenschaft, den View rotieren zu können.

```
01  on new me, dView, dModel, dRotateConstraints
02    ancestor = new(script "3DCTRLRotate", dView, dModel,\
                                        dRotateConstraints)

03    -- rotate ohne Zusatztaste
04    me.pRotateOption = [#none:#XZ, #shift:#XY]
05    return me
06  end

07  on mWhileDraggedAction me
08    if (integer(me.pView.mGetOrientation().yaw) = 0))  then
09      if not me.pModel.pVerschlossen then
10        me.pModel.mSchliessen()
11      end if
12    end if
13  end
```

In der Zeile 04 wird der Rotationsmodus festgelegt. Sie besagt, dass keine Zusatztaste (`#none`) gedrückt werden muss, um das Objekt in XZ-Ebene (im Koordinatensystem für Rotationen um Yaw und Roll) zu rotieren. Das gewährleistet also eine Rotation um die Y-Achse, ohne die Tastatur benutzen zu müssen. Bei jeder Rotationsbewegung

wird die Methode `mWhileDraggedAction()` aufgerufen. In ihr wird abgefragt, ob der Winkel der Tür 0 Grad beträgt, was bedeuten würde, die Tür ist geschlossen ist. In diesem Fall wird eine Meldung an das Modell geschickt (Zeile 10).

6.5 Anbindung der Viewobjekte an die 3D-Modelle

Die „Verbindung“ der Viewklassen mit den 3D-Modellen wurde in Abschnitt 5.2 aus Seite 81 beschrieben. Die Erzeugung der View-Objekte und 3D-Modelle findet schon bei der Darstellung des Viewports statt und kann durch den Entwickler nicht beeinflusst werden. Um die Objekte später referenzieren zu können, müssen sie als *Interactive* markiert werden. Sie werden dann in einer vom Entwickler benannten globalen Variablen gespeichert. Am Beispiel der Mikrowelle wird dem Mikrowellengrundkörper das Skript `Mkr3DView` angehängt und das Objekt über die globale Variable `mkrMaster` angesprochen. Die Namen der globalen Variablen sollten möglichst konsistent gewählt werden, um den Code leicht lesbar zu halten. Das an den Namen angehängte „Master“ erinnert den Entwickler daran, dass dieses Objekt selbst niemals im Labor zu sehen sein wird, sondern nur Klonierungen von diesem Objekt. An Unterobjekten braucht der Entwickler üblicherweise keine speziellen Variablennamen zu vergeben, da sie nie direkt referenziert werden müssen. Ihnen müssen jedoch die entsprechenden Skripte angehängt werden und sie müssen als interaktiv markiert werden.

6.6 Konfiguration der Szene

Neben der Erstellung der für den Anwender sichtbaren und interaktiven Objekte müssen weitere Szenenobjekte ausgerichtet und erstellt werden. Dazu gehören Räume, Kameras und Lichtquellen sowie virtuelle Arbeitsbereiche. Kameras und Lichtquellen können wie „normale“ 3D-Objekte behandelt werden. Ihnen können Skripte und globale Variablen zugewiesen werden. Neue Lichtquellen und Kameras werden direkt über die GUI von 3D DREAMS erstellt werden.

Für jeden Arbeitsbereich sollte ein virtueller Arbeitsraum definiert werden. Dafür wurde die Klasse `virtRoom3DView` implementiert, die in Abschnitt 5.6.3 beschrieben wurde. Die Begrenzungen dieses unsichtbaren Arbeitsraumes hängen von der Entfernung und vom Sichtwinkel der Kamera ab und müssen gegebenenfalls für jeden Arbeitsplatz spezifisch implementiert werden.

Für den Fall, dass sich das Labor aus mehrere Räumen zusammensetzt, müssen weitere Weltobjekte angelegt werden. Jedes Weltobjekt benötigt eine eigene Kamera, Lichtquellen und Arbeitsbereiche. Der Wechsel in einen Raum kann durch einfaches Aktivieren einer Kamera innerhalb dieses Raumes erfolgen.

6.7 Moduldefinitionen

Für jeden Versuch und Teilversuch muss ein Modul definiert werden. Im Vergleich zur Definition der Module für 2D-Versuche im GENLAB-Framework gibt es kleine Änderungen in der Definition für 3D-Versuche. Eine Beispieldefinition kann wie folgt angegeben werden (`classStandaloneTest3D`):

```
global schalter1Master, spitzeMaster, mkrMaster, abf1Master,\
                          becherMaster -- notwendig fuer 3DD
property ancestor

-- GUIComponents
property anleitungView

-- LabObjects
property schalter1View, schalter2View, spitzeView, mkrView, \
                               spitze1View, abf1View, mbeView

global gWindowManager, gSubstanzHandler, gLabmanager

on new me, dAnleitungType
  ancestor = new(script "classModuleStandalone3D")
  return me
end

on mCreateLaborComponents3D me
  me.schalter1View = schalter1Master.mNew()
  me.schalter2View = schalter1Master.mNew()
  me.spitzeView = spitzeMaster.mNew()
  me.spitze1View = spitzeMaster.mNew()
  me.mkrView = mkrMaster.mNew()
  me.mbeView = becherMaster.mNew()
  me.abf1View = abf1Master.mNew()

  -- korrekte Darstellung nur nach einem "go to the frame"\
  -- und "updateStage"
  go to the frame
  updateStage

  me.pLabComponentList = [schalter1View, schalter2View,\
       spitzeView, mkrView, spitze1View, abf1View, mbeView]
end

on mSetLaborComponentsLocation3D me
  abf1View.mSetLoc([#x:0.0, #y:0.0, #z:0])

  schalter1View.mSetLoc([#x: -56.8044, #y: 12.5001, #z: 6.9211])
  schalter1View.mSetOrientation(\
```

```
            [#yaw: 333.0000, #pitch: 0.0000, #roll: 0.0000])

  schalter2View.mSetLoc([#x:0.0, #y:7.5, #z:20])
  schalter2View.mSetOrientation(\
            [#yaw: 20.0, #pitch: 0.0000, #roll: 0.0000])

  spitzeView.mSetLoc([#x: -7.1866, #y: 7.0247, #z: -23.6050])
  spitzeView.mSetOrientation(\
            [#yaw: 32.0000, #pitch: 0.0000, #roll: 0.0000])

  spitze1View.mSetLoc([#x: -15.0000, #y: 15.6469, #z: -20.0000])
  spitze1View.mSetOrientation(\
            [#yaw: 230.5555, #pitch: 90.0, #roll: 186.0690])

  mkrView.mSetLoc([#x: 64.6403, #y: 22.5019, #z: 25.3558])
  mkrView.mSetOrientation(\
            [#yaw: 30.5000, #pitch: 0.0000, #roll: 0.0000])
end

on mInitialPrepareActions3D me
  mkrView.pModel.mOeffnen()
  mkrView.mEinfuegen(mbeView)
  mkrView.pModel.mNextCountDownStart()
  mkrView.pModel.mNextCountDownStart()
end
```

Zur Erzeugung der Komponenten werden die globalen Variablen, die die Masterobjekte referenzieren, verwendet. In obiger Definition werden zwei Schalter, zwei Spitzen, die Mikrowelle, ein Messbecher und ein Arbeitsbereich erstellt. In der Methode `mSetLaborComponentsLocation3D` werden alle Komponenten ausgerichtet und positioniert. `mInitialPrepareActions3D()` sorgt dafür, dass die Mikrowelle geöffnet wird, der Meßbecher hereingestellt und die Zeit der Mikrowelle auf die zweite Stufe gestellt wird. Diese Versuchsaufstellung wurde für die Abbildung 5.6 verwendet.

Der Versuch kann gestartet werden, indem folgende Anweisung an den Labormanager geschickt wird:

```
mGotoStandAlone("classStandaloneTest3D")
```

6.8 Distribution

Zur Distribution von DIRECTOR-Filmen bietet DIRECTOR die Möglichkeit, an einen Projektor oder einen SHOCKWAVE-Film zu generieren. Dazu müssen alle 3D-Modell-Informationen (Geometrie, Material und Texturen) bereits in Castmember importiert worden sein (siehe Abschnitt 5.2 auf Seite 80). Der Projektor oder SHOCKWAVE-Film

arbeitet nicht mit dem Authoring-Xtra von 3D DREAMS zusammen, sondern nur mit dem von 3D DREAMS mitgelieferten Xtra 3DDPLAY.

7 Bewertung und Ausblick

In der vorliegenden Arbeit wurden Konzepte zur direkten Integration dreidimensionaler Modelle in virtuelle Labore entworfen und implementiert.

Ausgehend von einer detaillierten Analyse der Anforderungen an eine solche Integration und den zu berücksichtigen Problemen wurden Entwicklungswerkzeuge geprüft und ausgewählt, welche die Erstellung virtueller Labore mit dreidimensionalen Modellen unterstützen. Der anschließende Entwurf der 3D-Visualisierung trennt die Berechnung und Darstellung einer 3D-Szene, die Kollisionserkennung und die vom Labor- bzw. Versuchskontext abhängigen Bestandteile in eigenständige, unabhängige Komponenten. Für den Daten- und Kontrollfluss zwischen diesen Komponenten wurden Schnittstellenklassen entworfen. Dadurch sind sie leicht austauschbar und können durch andere ersetzt werden, was gerade wegen der zur Zeit stetigen Verbesserungen und Neuentwicklungen im Bereich der interaktiven 3D-Welten von Vorteil ist. Die Konzeption der 3D-Visualisierung für virtuelle Labore erfolgte auf Basis des GENLAB-Frameworks. Dieses wurde um Klassen erweitert, die eine dynamische Erzeugung, eine interaktive Manipulation und Interaktionen zwischen den 3D-Objekten ermöglichen. Die Modelle, also die Klassen, welche die internen Zustandseigenschaften der Laborkomponenten implementieren, sowie die Klassen zur Verwaltung und Erstellung der Versuchsabläufe können direkt aus dem GENLAB-Framework verwendet werden.

Die Erstellung eines Labors kann nach einem klar definierten nach Personen gegliederten Arbeitsablauf erfolgen. Die Entwicklung des Prototypen zeigte insbesondere einen im Vergleich zum 2D-Labor geringen Arbeitsaufwand zur Erstellung eines interaktiven Gerätes. Das Zerlegen und wieder Zusammensetzen einer Grafik entfällt, genau wie die Generierung von Standflächensprites zur Simulation der Tiefe eines Objektes. Zudem können vom 3D-Modell mehrere Instanzen erstellt, je nach Bedarf mit interaktiven Fähigkeiten ausgestattet und in beliebiger Perspektive dargestellt werden. Das 3D-Modell lässt sich sowohl in der Experimentierumgebung als auch in der Rahmenhandlung des Labors mehrfach einsetzen. Eine Aktualisierung des 3D-Modells spiegelt sich in allen Repräsentationen wider, so dass die Darstellungen immer konsistent zueinander sind. Zur einfacheren und schnelleren Bedienung des Labors kann der Entwickler die Manipulationsmöglichkeiten und die Art, wie diese durchgeführt werden können, für jedes einzelne 3D-Objekt flexibel festlegen. Das prototypische Labor zeigt auf üblichen Multimedia-Rechnern eine gute Performance bei der Interaktion mit den Geräten.

Aus der 3D-Visualisierung ergeben sich weitere Möglichkeiten für die Entwicklung

und Darstellung virtueller Labore, die auf Basis der hier vorgestellten Konzepte und Implementierung erarbeitet werden können:

- **Navigation.** Die 3D-Visualisierung bietet den Vorteil, dass alle Objekte aus einer beliebigen Perspektive dargestellt werden können. Diese Eigenschaft kann bei einer Navigation ausgenutzt werden, indem zum Navigieren beispielsweise Kamerafahrten verwendet werden.

- **Layoutmanager.** In GENLAB hat der Anwender die Möglichkeit, zu jedem Zeitpunkt eine Animation zu starten, welche die Versuchsdurchführung teilweise vorführt. Dabei kann es zu Veränderungen der Positionen der Laborkomponenten kommen, z. B. wenn ein Objekt aus einem Behälter genommen wird. Das Objekt muss an irgendeiner Position auf der Arbeitsfläche abgestellt werden, die noch nicht belegt ist. Anhand der Dimensionen der 3D-Objekte kann ein Layoutmanager berechnen, wo noch genügend Platz auf der Arbeitsfläche für das herausgenommen Objekt zur Verfügung steht. Der Layoutmanager könnte ebenfalls einen Weg berechnen, damit ein Objekt von Position A nach Position B bewegt werden kann, ohne andere Objekte zu berühren. Eine weitere insbesondere für die mechanische Physik denkbare Aufgabe ist die Anordnung und Ausrichtung von Geräten untereinander in einer bestimmten Weise.

- **3D-Editor.** Zur einfachen visuellen Erstellung von interaktiven Versuchskomponenten kann ein 3D-Editor hilfreich sein. Zu den Aufgaben des Editors können gehören:

  - Fertiggestellte „Behaviours“ per Drag & Drop auf einzelne Elemente des 3D-Modells ziehen.
  - Platzieren von Transformationsknoten.
  - Eingabe von Rotations- und Bewegungsbereichen durch Dialogboxen oder direkt durch visuelle Interaktion.
  - Festlegen durch welche Eingabe eine Manipulation durchgeführt werden kann.
  - Auswahl der Objekte, die mit diesem Objekt interagieren dürfen.
  - Initiale Positionierung und Ausrichtung.

- **Schattenwurf.** Zur leichteren Positionierung der dreidimensionalen Objekte durch den Anwender kann ein Schattenwurf dienen. Dadurch erhält er beim Bewegen stets einen Eindruck, wo das Objekt fallengelassen werden würde, wenn er die Maustaste loslässt.

Anhang

A Listing zur Erstellung einer 3D-Szene mit dem Mathengine-Toolkit

```
/*
  Copyright MathEngine PLC 2000
  www.mathengine.com   $Name: AUTO_TEMP_MeTKMakeRelease_1_299 $   $Id: BallHitsWall1.c,v
 1.12 2000/08/31 13:47:52 lyndonh Exp $
*/
/*
  Overview:   BallHitsWall1 uses the Collision Toolkit alone, without the Dynamics
  Toolkit. The scenario: a ball is thrown at a wall, bounces off the
  wall, bounces off the floor, and then exits the scene. The motion of
  the ball is not being driven by the dynamics engine, rather, it is
  scripted by hand.   Note: There is no gravity in this simulation.
*/ #include <math.h>
#include <stdio.h>
#include <stdlib.h> #include "McdFrame.h"
#include "McdPrimitives.h"
#include "MeMath.h"
#include "MeViewer.h"
#include "MeMemory.h" /* define to get cube collision */
#if 0
#define USE_CUBE
#endif /*
  Globals
*/
#define MAX_BODIES 10 /*
  MeMemory
*/
struct MeMemoryOptions opts; /*
  render context
*/
RRender *rc; /*
  Worlds and Spaces
*/
McdSpaceID space; /*
  Collision reps
*/
McdModelID groundCM;
/*
  collision model geometry representations
*/
McdGeometryID plane_prim, box_prim, box_prim1, ball_prim; /*
  Graphics reps
*/
RGraphicsDescription *groundG;
RGraphicsDescription *boxG;
RGraphicsDescription *ballG;
/*
  time step
*/ MeReal step = 0.02f;
/*
  ball radius
*/
MeReal radius = 0.5;
/*
  wall dimensions
*/
MeReal wallDims[3] = { 0.5f, 2.0f, 5.0f };
/*
  floor dimensions
*/
MeReal floorDims[3] = { 6.0f, 0.05f, 5.2f }; /*
  Colors
*/
float orange[3] = { 1.0f, 0.4f, 0.0f };
float green[3] = { 0.0f, 1.0f, 0.0f };
float blue[3] = { 0.0f, 0.598f, 0.797f }; /*
  rotate ground plane collision model 90 about x-axis
*/
MeReal groundTransform[4][4] =
    { {1, 0, 0, 0}, {0, 0, -1, 0}, {0, 1, 0, 0}, {0, 0, 0, 1} }; /*
  help text
*/
```

```
#ifndef PS2
char *help[2] =
{
    "Press enter to reset",
    "Press Spacebar to shoot"
};
#else
char *help[2] =
{
    "Press Start to reset",
    "Press Square to shoot"
};
#endif /*
  Structure that holds information about an object
*/
typedef struct Object
{
    McdModelID m_object;
    MeMatrix4 m_TM;
    MeReal *m_position;
    MeReal m_velocity[3];
}
Object; /*
  Object instances
*/
Object box;
Object ball; /*
  Set vector 'a' to be the sum of vector 'b' and the product of scalar
  'x' and vector 'c'. This is mainly used to drive the motion of the
  ball.
*/
void MultiplyAddVec(MeReal *a, const MeReal *b, const MeReal x,
    const MeReal *c)
{
    int i;     for (i = 0; i < 3; i++)
        a[i] = b[i] + x * c[i];
} /*
  Shoot the ball
*/
void shoot(void)
{
    ball.m_TM[3][0] = 0.0;
    ball.m_TM[3][1] = 4.0;
    ball.m_TM[3][2] = 0.0;
    ball.m_velocity[0] = 2.0;
    ball.m_velocity[1] = -2.0;
} /*
  Reset the scene
*/
void reset(void)
{
    MeMatrix4MakeIdentityTM(ball.m_TM);
    ball.m_TM[3][0] = 0.0;
    ball.m_TM[3][1] = 4.0;
    ball.m_TM[3][2] = 0.0;
    ball.m_velocity[0] = 0.0;
    ball.m_velocity[1] = 0.0;
    ball.m_velocity[2] = 0.0;     MeMatrix4MakeIdentityTM(box.m_TM);
    box.m_TM[3][0] = 3;
    box.m_TM[3][1] = wallDims[1];
    box.m_TM[3][2] = 0;
} /*
  Update the positions of the objects
*/
void move()
{
    MultiplyAddVec(box.m_position, box.m_position, step,
        box.m_velocity);
    MultiplyAddVec(ball.m_position, ball.m_position, step,
        ball.m_velocity);
} /*
```

```
  Handle collision for a given list of McdModelPairLinks. If a
  collision is is detected, change velocity of the ball, given the
  contact information
*/ void handleCollision(McdModelPair ** pairList, int NPairs)
{
    static McdContact contacts[10];
    float k;
    int i;
    McdIntersectResult result;     result.maxContactCount = 10;
    result.contacts = contacts;     for (i = 0; i < NPairs; i++)
    {
        McdModelPair *pair = pairList[i];         /*
          get contact info
        */
        McdIntersect(pair, &result);
        printf("\nNumber of contacts: %d", result.contactCount);
        /*
          calculate change of velocity (along normal) scalar
        */
        k = -2 * MeVector3Dot(ball.m_velocity, result.normal);
        /*
          change velocity
        */
        MultiplyAddVec(ball.m_velocity, ball.m_velocity, k,
            result.normal);
    }
} /*
  This function is called every frame by the renderer.
*/
void tick(RRender * rc)
{
    static McdModelPair* pairs[100];
    int NPairs;     McdSpaceHelloPairIterator helloIter;
    McdSpaceStayingPairIterator stayingIter;
/*    McdSpaceGoodbyePairIterator goodbyeIter;*/     RStopTimer(kRRender);
    RStartTimer(kRDynamics);
    move();
    RStopTimer(kRDynamics);
    RStartTimer(kRCollision);
    /*
      evolve space
    */
    McdSpaceUpdate(space);
    McdSpaceHelloPairIteratorInit(space, &helloIter);
    NPairs = McdSpaceGetHelloPairs(space, &helloIter, pairs, 100);
    handleCollision(pairs, NPairs);
    McdSpaceStayingPairIteratorInit(space,&stayingIter);
    NPairs = McdSpaceGetStayingPairs(space, &stayingIter, pairs, 100);
    handleCollision(pairs, NPairs);
    RStopTimer(kRCollision);
    RStartTimer(kRRender);
} void cleanup(void)
{
    /*
      memory release. Collision objects must be explicitly deleted
      by the user.
    */
    RDeleteRenderContext(rc);
    McdGeometryDestroy(plane_prim);
    McdGeometryDestroy(box_prim1);
#ifdef USE_CUBE
    McdGeometryDestroy(box_prim);
    McdModelDestroy(box.m_object);
#else
    McdGeometryDestroy(ball_prim);
    McdModelDestroy(ball.m_object);
#endif
    McdModelDestroy(groundCM);
    McdSpaceDestroy(space);
    McdTerm();
} /*
```

```
  Main Routine
*/
int main(int argc, const char **argv)
{
    RRenderType render;     /*
      Initialize memory
    */     MeMemorySetDefaults(&opts);
    (*MeMemoryAPI.setOptions)(&opts);     /*
      Initialize renderer
    */
    render = RParseRenderType(&argc, &argv);     rc = RNewRenderContext(render, kRQualit
ySmooth);     /*
      Initialize toolkit
    */     McdInit(McdPrimitivesGetTypeCount());     McdPrimitivesRegisterTypes();
    McdPrimitivesRegisterInteractions();     space =
        McdSpaceAxisSortCreate(McdAllAxes, MAX_BODIES, 2 * MAX_BODIES);
    McdPairHandlerRegisterSpace(space);
    /*
      ground plane
    */
    groundG =
        RCreateCube(rc, 2 * floorDims[0], 2 * floorDims[1],
        2 * floorDims[2], blue, 0);
    RSetTexture(groundG, "checkerboard");
    /*
      top of plane at y=0
    */
    groundG->m_matrixArray[13] = -floorDims[1];
    plane_prim = McdPlaneCreate();
    groundCM = McdModelCreate(plane_prim);
    McdSpaceInsertModel(space, groundCM);
    McdModelSetTransformPtr(groundCM, groundTransform);
    /*
      optimize
    */
    /* Inform the system that groundCM 's transform will not change value*/
    McdModelFreezeInSpace(groundCM);     /*
      box
    */
    box_prim1 = McdBoxCreate(2*wallDims[0], 2*wallDims[1], 2*wallDims[2]);
    MeMatrix4MakeIdentityTM(box.m_TM);
    box.m_TM[3][0] = 3;
    /*
      raise above floor
    */
    box.m_TM[3][1] = wallDims[1];
    box.m_TM[3][2] = 0;
    box.m_position = &box.m_TM[3][0];
    box.m_object = McdModelCreate(box_prim1);
    McdModelSetTransformPtr(box.m_object, box.m_TM);
    McdSpaceInsertModel(space, box.m_object);
    boxG =
        RCreateCube(rc, 2 * wallDims[0], 2 * wallDims[1],
        2 * wallDims[2], blue,
        (MeReal *) McdModelGetTransformPtr(box.m_object));
#ifdef USE_CUBE
    box_prim = McdBoxCreate(2*radius, 2*radius, 2*radius);
#else
    ball_prim = McdSphereCreate(radius);
#endif     MeMatrix4MakeIdentityTM(ball.m_TM);
    ball.m_TM[3][0] = 0;
    ball.m_TM[3][1] = 4;
    ball.m_TM[3][2] = 0;
    ball.m_position = &ball.m_TM[3][0]; #ifdef USE_CUBE
    ball.m_object = McdModelCreate(box_prim);
#else
    ball.m_object = McdModelCreate(ball_prim);
#endif     McdModelSetTransformPtr(ball.m_object, ball.m_TM);
    McdSpaceInsertModel(space, ball.m_object); #ifdef USE_CUBE
    ballG = RCreateCube(rc, 2*radius, 2*radius, 2*radius, orange,
        (MeReal*)McdModelGetTransformPtr(ball.m_object));
```

```
#else
    ballG = RCreateSphere(rc, radius, orange,
        (MeReal *) McdModelGetTransformPtr(ball.m_object));
    RSetTexture(ballG, "ME_ball3");
#endif     /*
      Build Space
    */
    McdSpaceBuild(space);     /*
      Keyboard callbacks
    */ #ifndef PS2
    RUseKey('\r', reset);
    RUseKey(' ', shoot);
#else
    RUsePadKey(PADstart, reset);
    RUsePadKey(PADsquare, shoot);
#endif     /*
      On screen help text
    */
    RCreateUserHelp(help, 2);
    rc->m_title = "BallHitsWall1 tutorial";     /*
      cleanup after simulation
    */
#ifndef PS2
    atexit(cleanup);
#endif
    /*
      run the simulation loop
    */
    RRun(rc, tick);     return 0;
}
```

Glossar

3D-Modell: Datenstruktur zur Speicherung dreidimensionaler Abbilder. Die Speicherung erfolgt vektororientiert.

API: Abkürzung für: Application Programming Interface. ↑Schnittstelle für den Zugriff auf Funktionen und Daten eines Programms.

Autorensystem: Software-Entwicklungswerkzeug für ↑interaktive multimediale Anwendungen mit Hilfe der ↑visuellen Programmierung.

Beleuchtungsmodell: Nachbildung realer Beleuchtungsarten in Computerprogrammen

Beschreibungssprache: Sprache zur Definition von logischen Bestandteilen eines Dokumentes oder einer Struktur, die durch ein Programm interpretiert werden können.

Bibliothek: Eine Kollektion von Funktionen oder / und Klassen, die zu einem Programm gebunden werden können. Sie ist speziell für die Wiederverwendbarkeit organisiert und beschleunigen den Prozess der Softwareentwicklung.

Bitmap: Eine Bitmap ist eine Datenstruktur, die für jedes einzelne Pixel eines Bildes einen Farbwert speichert.

Bounding Box: Einfache Approximation eines geometrischen Objektes durch ein Rechteck oder Quader. Im allgemeinen ist es minimal, also das kleinste Rechteck oder Quader, welches das Objekt noch enthält, und achsenparallel.

Cast: Kollektion mehrerer ↑Castmember.

Castmember: Struktur in der in DIRECTOR verschiedene Medientypen gespeichert werden können.

Contact Determination: Ermittlung der bei einer ↑Kollision beteiligen Polygone eines Kollisionsobjektes.

Controller: Im Kontext des ↑MVC-Musters: Empfängt und reagiert auf Eingaben vom Anwender.

DirectX: ähnlich wie ↑OpenGL, aber von MICROSOFT entwickelt und nur für dessen Betriebssysteme verwendbar.

Entwicklungsumgebung: ↑IDE

Flächenkörper: Bezeichnung für ↑3D-Modelle, die aus ↑Polygonen konstruiert wurden.

Framework: Eine Kollektion von abstrakten und konkreten Klassen, die zur Erstellung themenverwandter Applikationen dienen.

fps: Abkürzung für: frames per second. Einheit für die Abspiel- oder Aufnahmegeschwindigkeit von bewegten Bildern.

Geometrie: *hier:* die Gestalt eines Objektes

HTML: Abkürzung für: HyperText Markup Language. ↑Beschreibungssprache für Texte. Besonderheit von ∼ ist die logische Formatierung von Textbestandteilen im Gegensatz zur absoluten Formatierung des Textlayouts. Zusätzliche Funktionen von ∼ sind die ↑Hyperlinks und das Einbinden multimedialer Bestandteile in den Text.

Hyperlink: Ein ∼ ist eine Verbindung zwischen einzelnen Medien. Sie werden vor allem im WWW vielfach benutzt, um auf andere WWW-Dokumente, die verwandte Informationen anbieten, zu verweisen.

Hypermedia: Ein Netzwerk verschiedener Medientypen, die über ↑Hyperlinks miteinander verknüpft sind.

IDE: Abkürzung für: Interactive Development Environment. Die ∼ ist ein System, das den Prozess des Schreibens von Software unterstützt.

Implementation/Implementierung: Umsetzung eines Software-Entwurfes in ein konkretes Programm.

Instanz: Ausprägung einer OO-Klasse. Ein konkretes Objekt der Klasse im Hauptspeicher.

Interaktion: wechselseitige Beziehung, gegenseitige Beeinflussung

Internet: Weltweiter Verbund von Computernetzen.

ISO: Kurzform für: International Organisation for Standardization. Organisation, welche die Entwicklung internationaler Standards vorantreibt.

Kamerametapher: Die ∼ wird oft zur ↑Navigation in dreidimensionalen Umgebungen verwendet. Der Navigator sieht die 3D-Szene aus der Richtung und Position der Kamera. Sie wird nur zur Spezifikation einer Sichtweise verwendet.

Kollision: Überschneidung oder Berührung zweier Objekte.

Kollisionsentdeckung: Prozess der Überprüfung von ↑Kollisionen.

Kollisionserkennung: ↑Kollisionsentdeckung.

Material: Das $\sim$ besteht aus Attributen zur Beschreibung des Erscheinungsbild eines Objektes (z. B. Farbe, ↑Transparenz, ↑Reflexion).

Manipulation: *hier:* Bedienung von Apparaten, Handhabung.

Meshobjekte: Anderes Wort für ↑Flächenkörper.

Model: Im Kontext des ↑MVC-Musters: Modelliert interne Zustandseigenschaften und Abläufe eines Objektes.

Modellierung: Prozess der Erstellung eines Abbildes von einem realen Objekt.

Multimedia: Eine Kodierung von Informationen in mindestens einem kontinuierlichen und einem diskreten Medium

MVC: Abkürzung für: Model-View-Controller. Entwurfsmuster zur Erstellung von Benutzungsoberflächen.

Navigation: Prozess, durch den Menschen ihre Bewegung unter Verwendung künstlicher Hilfen oder umgebungsspezifischer Hinweise kontrollieren.

NURB: Abkürzung für: Non Uniformed Rational B-Splines. $\sim$s können verwendet werden, um ↑3D-Modelle zu erstellen.

OFFIS: Abkürzung für: Oldenburger Forschungs- und Entwicklungsinstitut für Informatik-Werkzeuge und -Systeme.

OpenGL: Eine Schnittstelle zur Hardware, die Operationen zum Berechnen und Darstellen von 2D- und 3D-Objekten unterstützt.

Parallelprojektion: ↑Projektion bei die Projektoren parallel verlaufen und sich nicht wie bei der ↑Zentralprojektion in einem Zentrum treffen. Es findet keine ↑perspektivische Verkürzung statt.

Perpektive: Darstellung räumlicher Gebilde auf einer ebenen Fläche.

perspektivische Verkürzung: Effekt, bei dem weiter entfernte Objekte kleiner erscheinen als nahe Objekte.

Polygon: Regelmäßiges Vieleck.

Projektion: Abbildung eines Körpers auf einer zweidimensionalen Fläche.

Reflexion: Ablenkung eines Lichtstrahls beim Auftreffen auf ein Objekt.

Render-Engine: Programm, das alle Operationen zum ↑Rendern zur Verfügung stellt.

Rendern: Konvertierung von logischen objektbasierten Daten in eine ↑Bitmap

Schattierung: Berechnung von Farbwerten anhand eines ↑Beleuchtungsmodells.

Schnittstelle: Konvention zur Kommunikation zweier Softwarekomponente miteinander.

Spline: Mathematische Formulierung eine Kurve .

Sprite: Ein Bild mit beliebiger Form, das frei positionierbar ist.

synthetische Kamera: ↑Kameramethaper.

Textur: Eine ∼ ist eine ↑Bitmap, die auf ein 3D-Objekt gelegt werden kann und so das Erscheinungsbild des 3D-Objektes beeinflusst.

Texture-Mapping: Der Prozess eine ↑Textur auf ein 3D-Objekt zu platzieren.

Transformationsknoten: Objekt zur Organisation von 3D-Objekten in Gruppen.

Transparenz: Teilweise oder vollständige Unsichtbarkeit eines Objektes.

UML: Abkürzung für: Unified Modeling Language. Modellierungstechnik für objektorientierten Entwurf.

View: Im Kontext des ↑MVC-Musters: (audio-) visuelle Repräsentation der Zustände eines ↑Models.

Viewer: Programm zur ↑Visualisierung.

Viewport: Meistens rechteckiger Bereich, in dem die Ausgabe eines ↑Render-Prozesses stattfindet.

virtuell: Gegenteil von *real*.

virtuelles Labor: Modellhafte Nachbildung eines realen Labors in einem Computerprogramm.

Visualisierung: Prozess zur Darstellung von numerischen Informationen in grafischer Form.

visuelle Programmierung: Erstellung von Programmen mittels grafischer Techniken.

VRML: Abkürzung für: Virtual Reality Modeling Language. Sprache zur Beschreibung von dreidimensionalen Welten. ↑ISO-Standard.

Wireframe: Darstellung eines 3D-Objektes als Drahtgittermodell. Es werden nur die Polygonkanten gezeichnet.

Z-Order Sortierung: Überlagerung mehrer ↑Sprites, um einen Tiefeneidruck zu vermitteln

Zentralprojektion: ↑Projektion bei der sich die Projektoren in einem Zentrum schneiden. Sie hat den Effekt der ↑perspektivischen Verkürzung.

Literaturverzeichnis

[AAH+98] T. Aden, J. Appel, W. Heuten, S. Koopmann, T. Krüger, M. Lindner, M. Sachteleben, H. Schneider, C. Wichmann, D. Boles, P. Dawabi und M. Schlattmann. *Zwichenbericht Teil B der Projektgruppe Virtuelle naturwissenschaftlich-technische Labore im Internet*. Carl von Ossietzky Universität Oldenburg, FB Informatik, Interner Bericht IS 38, http://elvis.offis.uni-oldenburg.de/dokus.html, Juni 1998.

[AB99] Prof. Dr. Hans-Jürgen Appelrath und Dipl.-Inform. Dietrich Boles. *Methoden und Werkzeuge zur Entwicklung virtueller multimedialer naturwissenschaftlich-technischer Labore und Praktika*. OFFIS, Juli 1999. DFG-Antrag.

[Bal99] Cindy Ballreich. *Web3D Content Developer Survey*. http://www.vrml.org/TaskGroups/x3d/slides/ BallreichDevelopersSurveyMarch1999/index.htm, Mai 1999.

[BKH97] D.A. Bowman, D. Koller und L.F. Hodges. Travel in Immersive Virtual Environments: An Evaluation of Viewpoint Motion. In *Control Techniques, Proceedings of the Virtual Reality Annual International*, Seiten 45–52. Syposium 1997 (VRAIS), 1997.

[Bol98] Dietrich Boles. *Begleitbuch zur Vorlesung Multimedia-Systeme*. Carl von Ossietzky Universität Oldenburg, Fachbereich Informatik, Abteilung Informationssysteme, http://www-is.informatik.uni-oldenburg.de/ dibo/teaching/mm98/buch/main.html, Dezember 1998.

[Bou97] Paul Bourke. *Surface (polygonal) Simplification*. http://www.swin.edu.au/astronomy/pbourke/modelling/surfsimp/, July 1997.

[BRJ99] Grady Booch, Jim Rumbaugh und Ivar Jacobson. *Das UML-Benutzerhandbuch*. Addison-Wesley, 1999.

[BS98] Dietrich Boles und Marco Schlattmann. Multimedia-Autorensysteme - Grafisch-interaktive Werkzeuge zur Erstellung multimedialer Anwendungen. *Log In*, (1), 1998.

[Buß99] Frank Buß. Java erobert den Raum. *c't (Magazin für Computertechnik)*, (8):226–233, 1999.

[com] *FOLDOC Free On-Line Dictionary Of Computing*. http://www.foldoc.org/.

[Elf99] Sigurd Elfreich. *Entwicklung eines Java-basierten objektorientierten Frameworks für virtuelle naturwissenschaftliche Experimente im Internet*. Carl von Ossietzky Universität Oldenburg, Fachbereich Informatik, 1999. Diplomarbeit.

[Eps98] Bruce Epstein. *Lingo in a Nutshell*. O'Reilly & Associates Inc., 1998.

[FD94] James D. Foley und Andries van Dam. *Grundlagen der Computergraphik*. Addison Wesley, 1994.

[G+98] Erich Gamma et al. *Design Patterns*. Addison-Wesley, 1998.

[Gen00] GenLab, http://www.offis.de/projekte/dibo/projekt_mgp.htm. *Webseite des Multimedialen Gentechnischen Praktikums (GenLab)*, 2000.

[Heu99] Wilko Heuten. *Seminararbeit zur 3D - Modellierung*. Carl von Ossietzky Universität Oldenburg, FB Informatik, März 1999.

[KRSD98] Jörg Kloss, Robert Rockwell, Kornel Szabo und Martin Duchrow. *VRML97*. Addison-Wesley, 1998.

[LMCG] M. Lin, J. Manocha, J. Cohen und S. Gottschalk. Collision Detection: Algorithms and Applications. In Jean-Paul Laumond und M. Overmars, Hrsg., *Algorithms for Robotics Motion and Manipulation*, Seiten 129–142.

[O+97] OFFIS et al. *Entwicklung einer elektronischen multimedialen Arbeits- und Experimentierumgebung für die Bio- und Gentechnologie*, 1997. Vorhabensbeschreibung.

[OFF00] OFFIS, http://www.offis.de. *Webseite des Oldenburger Forschungs- und Entwicklungsinstituts für Informatik-Werkzeuge und -Systeme*, 2000.

[OW90] T. Ottmann und P. Widmayer. *Algorithmen und Datenstrukturen*. Number 70 in Reihe Informatik. BI-Wissenschaftsverlag, 1990.

[Tho95] Nigel Thomson. *Animationstechniken in Win32*. Microsoft Press, 1995.

[UNC99] UNC (University of North Carolina), http://www.cs.unc.edu/ geom/collide.html. *Interactive and Exact Collision Detection for Virtual and Simulated Environments*, 1999.

Index

3D-Modell
    Konstruktion, 106
3D-Objekt, 48

Animation, 10
Anzeigen, 98
API, 28
Arbeitsablauf, 105
Arbeitsbereich
    virtueller, 99
Autorensystem, 27, **28**

Bausteinkomponenten, 75, **96**
Beleuchtung, **13**
Beleuchtungsmodell, 12
Beschreibungssprache, **30**
Bibliothek, 27
BIT, 17
Bounding-Box, 19

Controller
    Erstellung, 111

Depth Cueing, 14
Depth-sort, 12
Distribution, 119
Drag & Drop, 67
Droptable, 93

Eingabegeräte, 21
Entwicklungsumgebung, 27
Erzeugung
    dynamische -, 82
Experimentierumgebung, 9

Flächenkörper, 108
Flatshading, 13
Framework, **27**

GenLab, 1
    Framework, **60, 61**
Gouraudshading, 13
Grafikformat, 25
Gruppenknoten, 48

Hardwareunterstützung, 16
Hierarchie, 24, **49**, 80, 110
    Manipulation, 52
Hyperlinks, 10

IDE, 27
Implementierungsumgebung, 79
Installation, 105
Interaktion, 10, 21, 55, 81
    Aufgaben, 17
    Behälter-, 94
    View-, 91

Java, 39, 44
    3D, 39, 42, 48

Kamera, 16, 57, 63
Kernlabor, 46
Kindknoten, 48
Kollisionserkennung, 4, **19**, 46, 56, 69, 92
Konfiguration, 116
Koordinatensystem
    Yaw, Pitch, Roll, 53

Labormanager, 63, 75
Layoutmanager, 122
Lingo, 1, **33**, 80

Manipulation, 81
Masterobjekt, 83
Material, 12
Materialeigenschaften, **15**
Mathengine, 40
Meshobjekt, 49
Model-View-Controller, **61**
Modellierungstechnik, 23, 108

Module, 62, 73
    Definition, 117
    Fertikeiten-, 62
    Standalone, 62
    Versuchs-, 62
MVC, **61**

Navigation, **20**, 122
NURBS, 108

OFFIS, 1
OpenGL, 16
Operation
    boolesche, 24, 108

Parallelprojektion, 16
Performance
    Steigerung der, 99
Perspektive, 6
Phongshading, 13
Polygon, 12, 14
    Reduktion, 107, 109
Positionierung, **18**
Primitive, 24, 63
Programmierumgebung, 27
Projektion, 12, **15**

Quicktime, 40

Radiosity, 13
Rahmenhandlung, 9, 105
Raytracing, 12, 13, 15
Reflexion, 12, **14**
Render-Engine, 46, 55
Rotation, **18**

Scanline, 12, 15
Schalter, 96
    Dreh-, 97
    Kipp-, 97
Schatten, 12, **15**, 122
Schattierung, 13, **13**
Schattierungsmodell, 12, 14
Schnittstellenlayer, 47, 51
Shockwave, 34
Sprite, 2
Standfläche, 4

Taste, 96
Taster, 96
Textur, 12, **15**
Trägerbeziehung, 91
Transformation, 53, 72
    Einschränkungen, 90
    interaktive, 87
    Knoten, 49, 53, 58
    Modus, 72, 89
Transparenz, 12, **14**

UML, 45
Universum, 48

Vererbung
    Einfach-, 51
    Mehrfach-, 50
Versuchskomponenten, 9
Video, 10
View
    Anbindung an 3D-Modelle, 116
    Erstellung, 111
Viewport, 60, 63
Virtuelles Labor, 9
VRML, 42, 48
    Viewer, 49

Wireframe-Darstellung, 100
Workflow, 105

z-Buffer, 12
Z-Order, 4, 6, 69
Zentralprojektion, 16

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen verwendet habe.

Oldenburg, den 30. April 2001

(Wilko Heuten)